/THEORY/IN/PRACTICE

软件开发者路线图：

从学徒到高手

Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman

Dave H. Hoover & Adewale Oshineye 著
Ward Cunningham 序
王江平 译

REILLY®
机械工业出版社
China Machine Press
HZ BOOKS
华章科技

Software Engineering

Apprenticeship Patterns

“了不起的作品！阅读本书就像置身于一部时间机器中，把我带回软件开发者职业生涯中的那些关键的学习时刻，那时，学习最佳的实践方法需要通过艰苦的方式，但是在我从学徒到高手的每一步上都有良师益友坐在旁边。我当然乐意把这本书介绍给大家。我多么希望14年前就拥有了这本书！”

CEO

作为一名软件开发者，你在奋力推进自己的推进自己的推进自己的推进自己的职业生涯的技术，取得成功需要的不仅仅是技术专长。为了术专长。为了术专长。为了术专长。为了增强专的学习技能。本书的全部内容都是关于如何修炼于如何修炼于如何修炼于如何修炼这些技wale Oshineye给出了数十种行为模式，来帮你提高业要的技能。来帮你提高。来帮你提高主要的

本书中的模式凝结了多年的调查研究、无数次的访谈以及来自O’Reilly在线论坛的反馈，可以解决程序员、管理员和设计者每天都会面对的困难情形。本书介绍的不只是经济方面的成功，学徒模式还把软件开发看成一种自我实现的途径。读一读这本书吧，它会帮你充分利用好自己的生命和职业生涯。

- 厌倦了自己的工作？去找一个玩具项目来帮你重拾解决问题的乐趣吧，这叫“培养激情”。
- 感觉要被新知识淹没了？做点以前做过的事情，重新探索一下自己熟悉的领域，然后通过“以退为进”再次前进。
- 学习停滞了？那就去寻找一支由富有经验和才能的开发者组成的团队，暂时呆在里面“只求最差”。

Dave H. Hoover：Obtiva首席技师，喜欢在开发软件的同时培养软件开发者，他的专长是向企业家们交付项目。

Adewale Oshineye：软件工程师，从事过包括电子零售商销售网点系统、投资银行交易系统在内的各种大型项目开发。

客服热线：(010) 88378991，88361066
购书热线：(010) 68326294，88379649，68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

O'Reilly Media, Inc.授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®
www.oreilly.com

ISBN 978-7-111-31006-8
9 787111 310068

定价：35.00元

软件开发者路线图：从学徒到高手

David H. Hoover & Adewale Oshineye 著

Ward Cunningham 序

王江平 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

软件开发者路线图：从学徒到高手/（美）胡佛（Hoover, D. H.）等著；王江平译. —北京：机械工业出版社，2010.7

书名原文：Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman

ISBN 978-7-111-31006-8

I. 软… II. ①胡… ②王… III. 软件开发 IV. TP311.52

中国版本图书馆CIP数据核字（2010）第113005号

北京市版权局著作权合同登记

图字：01-2009-7020号

书　　名/　软件开发者路线图：从学徒到高手

书　　号/　ISBN 978-7-111-31006-8

责任编辑/　迟振春

封面设计/　Mark Paglietti，张健

出版发行/　机械工业出版社

地　　址/　北京市西城区百万庄大街22号（邮政编码100037）

印　　刷/　北京京师印务有限公司

开　　本/　170毫米×242毫米　16开本　12.5印张

版　　次/　2010年8月第1版 第1次印刷

定　　价/　35.00元（册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991; 88361066

购书热线：(010) 68326294; 88379649; 68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc.授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的*The Whole Internet User' Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

作者简介

Dave Hoover：Obtiva首席技师，喜欢在开发软件的同时培养软件开发者，他的专长是向企业家们交付项目。

Adewale Oshineye：软件工程师，从事过包括电子零售商销售网点系统、投资银行交易系统在内的各种大型项目开发。

目录

序 1

前言 5

软件工艺宣言 19

第1章 绪论 21

什么是软件技能 25

学徒期是什么 31

学徒模式是什么 32

模式来自哪里 33

下一步做什么 33

第2章 空杯心态 35

入门语言 38

白色腰带 45

释放激情 49

具体技能 51

暴露无知 54

正视无知 57

深水区域 59

以退为进 62
总结 64

第3章 走过漫漫长路 67
漫漫长路 69
技重于艺 71
持续动力 74
培养激情 77
自定路线 80
使用头衔 84
坚守阵地 85
另辟蹊径 87
总结 89

第4章 准确的自我评估 93
只求最差 94
找人指导 98
同道中人 101
密切交往 104
打扫地面 107
总结 109

第5章 恒久学习 113
提高带宽 114
不断实践 118
质脆玩具 121
使用源码 124
且行且思 128
记录所学 131
分享所学 133
建立馈路 136

学会失败 139
总结 140

第6章 安排你的课程 143
阅读列表 144
坚持阅读 147
钻研名著 148
深入挖掘 150
常用工具 155
总结 158

第7章 结束语 161

附录A 模式列表 167

附录B 一次学徒培训的号召 171

附录C 回顾Obtiva学徒训练项目的第一年 175

附录D 在线资源 179

参考文献 181

序

25年前，我和Kent Beck坐在Tektronix技术中心的自助餐厅里，考虑着我们对Smalltalk-80的深入接触会给世界带来怎样的影响。

我对Kent说：真的不必担心。如果我们能做任何事情，我们会做什么呢？

“我想改变人们看待程序设计的方式。”Kent说。我表示赞同。那时，我们都认为行业发展进入了错误的方向，而我们都想逆转它。而且，令人惊奇的是，我们做到了。

我那时在自助餐厅中使用的这项技巧——“真的不必担心”这部分——是最早从我大学的指导老师那里学到的一种模式。他把该模式用在我身上，正如我把它用在Kent身上。这种行为我现在把它归结为一种模式，它使我和Kent敢于想象更远的目标，若非如此，那些目标看起来显得盲目。而一旦想到了，我们的目标就更容易实现了。

我把这种思考的技巧称为模式，因为它解决了我们经常遇到的一个问题：我们潜意识里会压抑自己的雄心壮志。本书中全是类似这样的技巧，它们用于解决各种不同的问题。我们说：模式解决问题。“真的不必担心”帮我和Kent解决了一个问题。它使我们真正去考虑自己本来就有的更大的想法，而且使我们克服了习惯性的自我压抑。

或许你也尝试过“真的不必担心”这一模式。如果没有，不妨试一试。

最强大的模式是那些可以反复运用并取得成功的模式。模式并不一定非要新颖才有用。事实上，不新的模式才更好。另外，仅仅知道几个成型的模式名字也没有太大帮助。确定出一种模式，以后你每次谈论它的时候就不必重述整个故事。

快速翻阅这本书，你将看到很多模式，其中许多都会让你觉得熟悉。对任何一个，你可能都会说“我已知道这种模式了”——可能事实真的如此。然而，即使相关解决方案是常识性的，书中的模式仍然能提供两种帮助你的方法。

第一，书中的模式更加完整。它们已经被研究、定性、归类并解释过。每一种模式都会带给你意外的收获。尽情享受这种收获吧，它会帮助你让已知的模式变得更加强大。

第二，这里的模式都是相互关联的。每一种模式都通向其他的多种模式。当你发现一种已经知晓的模式，沿着这些关联，你可以顺藤摸瓜找到其他一些尚未知晓或者从没想过可与之联合运用的模式。

我和Kent在Smalltalk-80中挖掘模式，我们找到了很多。我们将这些模式的概念讲给了同事们，并引发了一场小小的革命。我们改变了人们考虑程序设计的方式。从那时开始，人们写出了几十本有关模式以及如何使用它们的书籍。

我们的革命远未结束。慢慢地，模式术语越来越多，并成为敏捷（Agile）软件开发方法的基础。后来又出现了几十本书籍。

那现在为什么写这本书呢？好吧，我们的职业已经承载了过多的资源。关于我们的革命，有太多的信息可以获取，以至于没有人能完全吸收它们。然而，还是有些人努力做到了。他们消化了所有可以获得的建议，而且似乎总能在需要时信手拈来。他们是如何掌握到这种程度的呢？

本书的内容全都是教你如何掌握我们所处的复杂领域的模式。掌握不只是知晓。掌握是能帮你减轻负担的知晓。

举个例子，如果你记不住SUBSTR函数的参数顺序，可以到互联网上查一查。感谢上帝赐予我们互联网，它把我们的负担减轻了一点点。但是，当你运用本书中的模式，当你可以随时改善自己的工作，你会发现自己在写一种不同的代码，一种不依赖于了解SUBSTR参数顺序的代码。你将写出比SUBSTR飞得更高的程序，而这将大大减轻你的负担。

来自这次革命中的所有建议，除非能成为你的第二天性，否则就不会有太大的帮助。在软件领域，工艺的发展使我们认识到，让这些东西变成第二天性并不是我们的第二天性。这些模式是对这一过程的贡献，令人欢迎的贡献。

——Ward Cunningham

前言

不知而不知其不知者，愚者也——避之！
不知而知其不知者，惑者也——授之！
知之而不知其知之者，寐者也——醒之！
知之而知其知之者，觉者也——从之！
——Isabel Burton（1831—1896）女士在
《The Life of Captain Sir Richard F. Burton》
（Richard F. Burton上尉的一生）
一书中引用的阿拉伯谚语

本书目标

缺少经验的软件开发者常常面对进退两难的境地。为分享摆脱这类窘境的方法，我们写了这本书。我们指的不是技术上的窘境；本书中你不会找到任何Java设计模式或者Ruby on Rails诀窍。相反，我们所致力于解决的窘境是跟人有关的，是关于动机和士气的。作为专业软件开发领域的新人，你会面对一些艰难的决定，本书将帮助你做出这些决定。

读者对象

本书写给那些对软件开发有过一些体验并立志成为软件开发高手的人。你可能是一名Web应用开发者，或者是一名医疗设备程序员，或者正在为一家金融机构开发交易应用。你也可能刚从中学或大学毕业，并认为软件就是你的未来。

尽管本书是写给新人的，经验丰富的开发者仍可从本书内容中受益。有过几年开发经验的人，当从书中找到自己曾经面临过的窘境时，会不住地点头，而且还可能获得新的感悟，或至少获得一个新的词汇，用于描述他们自己使用或者建议给同行使用的方案。即使拥有十几年或者更长时间开发经验的人，特别是那些正在努力把握职业航向的人，也将会找到一些灵感和视角，用来抵御晋升到管理职位的诱惑。

写作过程

本书的构思始于2005年初，Stickyminds.com请Dave写关于软件工艺（Software Craftsmanship）的专栏。那时Dave认为自己是个（有经验的）学徒（apprentice），唯一让他写起来感觉舒服的主题就是学徒期（apprenticeship）。这使他开始考虑关于这个主题写些什么更好。大约就在那时，Dave读到一篇由软件开发者Chris Morris发表的网志[注1]，其中提到了吉他手Pat Metheny，而“只求最差”这一概念成了模式语言的种子。这粒种子很快从Dave的网志[注2]成长为Dave用来组织最初一些模式的个人wiki。最初的一些模式就是从Dave到那时（2000—2005）的职业生涯中提取出来的。

Dave知道，这些所谓的模式不能真正称为模式，除非它们确实是常见问题的一般解决办法。他开始通过三种方式从同行那里寻求反馈。第一，他开始在自己的网站上公开发表这些模式，通过公众点评的方式寻求反馈。第二，他开始访谈一些软件开发领域的思想领袖（主要通过电子邮件），并获得他们对最初几种模式的看法。第三，也是最重要的一种，Dave开始访谈一些经验较少的从业者，用他们新近的经验来检验这些模式。这些经验较少的从业者们还把一些Dave不曾遇到的或者在他的经验中不曾觉察到的模式介绍给他。就在这些关于学徒期的访问中Dave访问了Ade，经过双方同意，Ade加入到这个项目中并成为一名合著者。

注1：*http://clabs.org/blogki/index.cgi?page=/TheArts/BeTheWorst.*

注2：Red Squirrel Reflections（红松鼠沉思录），参见：*http://redsquirrel.com/cgi-bin/dave*。

我们（Dave和Ade）访谈了生活和工作在全世界很多地方的人们，从澳大利亚到印度再到瑞典。讨论的形式也五花八门，从LiveJournal网站的评论，到伦敦金融区中心一座美丽的被炸教堂遗址里进行的访问。

与此同时，其他一些人，如Laurent Bossavit、Daragh Farrell和Kraig Parkinson，他们勇敢地在各种练习班、研讨会和新手训练营中尝试使用这些模式。之后他们把收到的反馈传给了我们（Dave和Ade），而我们则竭尽全力地把这些反馈合并到我们的笔记中。

在2005年末的时候，我们在“程序的模式语言”（Pattern Languages of Programs，PLoP）研讨会[注3]上发起了一个主题小组。在PLoP上，我们得以将自己的工作呈现给那些经验丰富的模式作者（也称为牧羊人，shepherd）们，他们在模式的组织形式方面给了我们许多反馈，并用自己的程序开发经验测试了我们的论断。

大概在同一时间，来自O'Reilly Media的Mary Treseler联系了我们，提出了出版这些模式的建议，并鼓励我们继续写作。她帮我们做了一些编辑工作，两年之后我们就书的出版达成协议。在那段时间里，通过电子邮件、用户组和讨论会，甚至午餐时的谈话，我们同数不清的同行就这些模式进行交流，同时我们继续在*http://apprenticeship.oreilly.com*在线社区上寻求反馈。

最终的结果就是你手上的这本书。这是一部基于与从业者之间的无数次交流的作品，也是一部对已有的关于学习、最佳效率心理学，以及所有我们能找到的与掌握知识相关的文献进行广泛研究的作品。随着阅读的深入，你将看到，除了通常的软件领域的名人之外，我们还会引用外科医生、舞蹈编导还有哲学家的话。我们相信，向任何一门学科的高手学习，我们都能收获很多。

注3：*http://hillside.net/plop/2005/group1.html*.

内容编排

模式是对特定上下文中某个问题的可重现解决方案的命名描述。这种描述应使读者对问题产生足够深入的理解，使他们或者能将所述方案应用到自己的情景中，或者能确定某个方案对他们的情景不适合。

本书由较长的几章组成，每章都填满了一组彼此相关的模式。模式的名字以首字母大写（这是指英文原版书——编辑注）拼出（如“质脆玩具”，Breakable Toys），而相关的模式则频繁引用。每一章都把其中的模式编排在一起，并提供一个对本章主题的介绍，以及一个进行总结的小节。本书的绪论部分为模式语言打下基础，而结束语部分则审视了有关技巧、学徒期和掌握专业知识的“大图景”。

模式的形式

我们的模式形式与众不同。如果你读过其他关于模式的书籍，你会看到在本书中我们尝试了一些不同的东西。同大多数模式语言相比，我们关于抽象外因和约束的章节不多，相关的讨论也较少。选择这种形式是基于来自审稿人和PLoP研讨会的广泛反馈。基于这些反馈，我们相信这种简单的结构将使我们的模式语言更易被目标读者接受和理解。

我们的模式都包含一个“情景分析”、一个“问题描述”和一种“解决方法”，然后是一组单一或多重的行动。“情景分析”设好基调，“问题描述”则确定模式所解决的问题。“解决方法”通常从问题的一句话分析开始，接着深入到与运用方法有关的各类细节问题上，其间描述该模式与其他模式的关系，还有支持该模式的故事和文献。

在每种模式接近尾声的地方都有一个“行动指南”的小节，这一小节描述了你可以立即付诸实践的具体事情，如果你愿意体验模式效果的话。这些行动可用作实施的示例，它们提供了一些练习，你可以立即投入实践，这样就不用担心模式是否可用于你身边的情形。

任何模式都应包含给定情景中一大类问题的一揽子解决方案，记住这一点很重要。模式是用来修正以适应具体情形的，而不是用来生搬硬套

的。因此，如果一种模式不能准确适用于你的情形，或者“行动指南”小节中任何一条看起来都不合适，那么你应该尝试沿着我们提供的材料进一步外推，看看能否构造出一些有用的东西。

大多数模式都以一个“参考模式”的小节结尾，这一小节指出了相关模式的页码。这样你阅读的时候就可以不采用线性顺序，而是采用一种迂回的顺序，从而加深你对不同模式间关系的理解。

用法

> 模式语言将创造的力量赋予每一个使用它的人，使他们创造出无数独特的新构造，正如一般意义上的“语言”给予他们创造无数句子的力量一样。
>
> ——《The Timeless Way of Building》（建筑的永恒之道），第167页

我们这个项目的目标是创建一种模式语言，来帮你定义自己的学徒过程。我们不可能知道你所处的实际环境，因此一定要考虑每种模式的情景分析和问题描述来确定它是否适合你。这些模式之间是相互联系的，可以联合使用来创造更强有力的体验。比如，尽管“找人指导”本身就是一种优秀的、经过时间检验的模式，但将它跟“密切交往”结合起来会更强大。另一方面，“暴露无知”更加依赖一些支持模式，比如“正视无知”和“以退为进”，而且需要多用一点技巧才能正确使用。跟所有的模式语言一样，你应该小心，不能过度使用这些模式。不要为使用各种模式而寻找借口，而应针对具体情形遴选最恰当的组合。

你不一定要按从头到尾的顺序阅读书中的模式。当初Dave阅读Christopher Alexander的《A Pattern Language》（模式语言）时，就是从书的中间开始，并沿着模式之间的关联来阅读，这带来了一种更有趣的学习体验。你也许只想简单浏览一下各种模式的情景分析和问题描述来找到跟你的实际情形最切合的模式。按这种方式来扫描模式可以让你在大脑中安装一些触发器来应对未来情形，那时，你会突然发现某些模式可以应用。

本书的内容最初写在一个wiki上，它自然从未真正考虑过让人按线性顺序阅读。前面的模式会引用到后面的模式，反之亦然。这会带来一些挑战，而且需要你积极地投入到这些内容中。你可以像浏览一个网站那样浏览它，这会使你的注意力被各种有趣的链接分散，而永远不知道是否读完了所有的内容。但这种方法并没什么不好。

我们当然理解一些人更喜欢从头到尾地阅读。因此，我们在前面的几章中做过一些努力，使前向引用减到最少；前向引用是指一个模式会引用到书中在它之后出现的模式。

一些人会觉得他们需要把这本书通读两遍：首先，快速浏览，把所有的东西都记到脑子里；然后读二遍，把所有的东西连接起来。这种方法也不错。本书并不适合用作参考手册，而更像一本艺术家的资料合集——你可以时不时地翻阅它，从中汲取灵感。你甚至可能发明一些新的、我们从来没想过的使用本书的方法。尽管去尝试。本书跟现实世界中其他的东西一样：彼此的关联不见得一下子就能看清，而每次回顾时，你都会发现新的东西。

使用示例代码

本书的目的是帮你完成工作。一般来说，你可以将书中的代码用于你的程序或文档。除非你直接复制大段代码，否则无需联系我们并获得许可。举例来说，在程序中使用几段书中的代码无需许可。出售或发布含有O'Reilly书中示例的CD-ROM需经许可。引用本书中的叙述或示例代码来回答问题无需许可。将书中的大量代码合并到你的产品文档中则需获得许可。

虽然我们并不要求在引用本书的时候做版权归属声明，但如果你这样做了，我们会非常感激。版权归属声明通常包含标题、作者、出版者和ISBN。例如："*Apprenticeship Patterns* by David H. Hoover and Adewale Oshineye。Copyright 2010 David H. Hoover and Adewale Oshineye，978-0-596-51838-7."。

如果你觉得自己对书中示例代码的使用超出了正常范围，或者不符合以上描述的许可范围，请用电子邮件跟我们联系：*permissions@oreilly.com*。

我们的联系方式

你可以通过邮件地址*dave.hoover@gmail.com*联系Dave，或者访问他的主页，看看他最近又在做什么。

你可以通过邮件地址*ade@oshineye.com*联系Ade。通过他的网站可以连接到他的相册、作品和开放源代码。

有关本书的建议和问题请与出版商联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

本书也有相关的网页，我们在上面列出了勘误表、范例以及其他一些信息。你可以访问：

http://www.oreilly.com/catalog/9780596518387（英文版）
http://www.oreilly.com.cn/book.php?bn=978-7-111-31006-8（中文版）

对本书做出评论或者询问技术问题，请发送E-mail至：

bookquestions@oreilly.com

希望获得关于本书、会议、资源中心和O'Reilly网络的更多信息，请访问：

http://www.oreilly.com

http://www.oreilly.com.cn

Dave的致谢

我要从感谢那些给我第一次机会让我成为软件工程师的人开始。Irv Shapiro，Edventions公司的CEO，2000年4月在我们面谈结束时，他雇佣了我。那一年晚些时候，他把我介绍给了Steve Bunes（Edventions公司的CTO）和Risetime Technologies公司的CEO，他们两位都在我开始兴奋地步入Perl和程序设计的过程中指导了我。2001年4月，当Edventions跟许多其他的“dot-com”创业公司一样死去时，Steve在美国医学会（American Medical Association，AMA）为我说了好多美言，在那里我度过了接下来的三年，从互联网炸弹（dot-bomb）的灾难中挺了过来[译注1]。我要重复一下在Irv雇佣我的那个晚上，我向他敬酒时说过的话：“感谢你，Irv，感谢你愿意在我这个异教徒身上冒险一试。”

在美国医学会有两个人给了我第一次超越入门语言的机会。John Dynkowski从我身上发现了一些潜力，让我去做医学会最初的几个J2EE项目。他为这事费了不少的政治成本，我在他的部门工作的18个月里，他不断地鼓励我。Doug Fedorchak，我的直属上司，给我充分的自主权，让我向上层管理者推销极限编程（Extreme Programming）方法，并让我启动了医学会里的第一个极限编程项目。感谢你们，John和Doug，是你们允许像我这样经验不足却又热情洋溢的程序员来尝试自己的思想，并在你们的组织里“兴风作浪”。

如果要我说出一个对我和我的职业旅程产生了最大影响的人，那他就是

译注1：当年dot-com泡沫破裂被人戏称为dot-bombs。注意在英语中，com和bomb的发音仅仅开口时的辅音不同，一个是b，一个是c，再考虑到“bomb”是炸弹的意思，就会觉得这是很有意思的双关语。

Wyatt Sutherland。2002年我在芝加哥敏捷开发者团体（Chicago Agile Developers group，ChAD）中遇见Wyatt，那时他是团体的负责人。我向Wyatt提议让我做他的“学徒”，他同意定期跟我会面，共进早餐和午餐。他是一名四处奔波的敏捷开发顾问、一所本地大学的乐队指挥，还是四个孩子的父亲。尽管如此，他还是做到了与我的定期会面。谢谢你，Wyatt，谢谢你那些年对我的指导。这是一份无价的礼物，让我有信心离开美国医学会，并立志到Object Mentor或ThoughtWorks这样的公司去工作。

我还要感谢我的前雇主ThoughtWorks，是她让我接触了一大群有兴趣为本书献计献策的人，尤其是我的合著者Adewale Oshineye。感谢你，ThoughtWorks的首席科学家Martin Fowler[译注2]，感谢你抽时间和我在一起分享你关于写作的见解。2005年，Obie Fernandez邀请我参加亚特兰大敏捷开发会议并作关于学徒模式的报告，ThoughtWorks慷慨地为我支付了路费。感谢你，Obie，感谢你在奥本冈时在我们共同的项目中给予我的支持与鼓励，感谢你邀请我参加亚特兰大敏捷开发会议，感谢你在我错过了回家的航班时让我睡在你的住所。感谢你，我的朋友Laurent Bossavit，你在2005年法国极限编程会议上介绍了我们的模式，并帮我把会议抄本翻译成英语。感谢你，Daragh Farrell，感谢你在2005年悉尼Geeknight会议上介绍了我们的模式，并将讨论的视频发给了我。感谢你，Linda Rising，感谢你邀请我和Ade参加了2005年的“程序的模式语言”研讨会，在那里我们收到了大量重要的反馈，而且第一次（迄今为止也是唯一一次）有机会见了面（再一次感谢ThoughtWorks让Ade从伦敦飞到芝加哥来参加这次研讨会）。

在我刚开始研究这些模式时，我接触了很多软件开发社区的知名人士。他们付出宝贵的时间通过电话或电子邮件与我交流，基于他们数十年的经验提供反馈和建议。我非常感激Ken Auer、Jerry Weinberg、Norm Kerth、Ron Jeffries、Linda Rising、Dave Astels，还有Pete McBreen，

译注2：Martin Fowler，企业级软件领域的著名作者、演讲人和咨询师，著有《Refactoring》、《UML Distilled》及《Analysis Patterns》等软件领域的名著，几乎每本都是畅销书。

感谢他们付出宝贵的时间在我写作过程中给予指导。同时，我（还有后来的Ade）也接触了数十名经验并不那么丰富的人（就像我一样），从他们那里寻求一些关于模式的输入，并从他们的故事中挖掘具有普适价值的主题。

非常感谢以下人士：Adam Williams、Chris McMahon、Daragh Farrell、Desi McAdam、Elizabeth Keogh、Emerson Clarke、Jake Scruggs、Kragen Sitaker、Ivan Moore、Joe Walnes、Jonathan Weiss、Kent Schnaith、Marten Gustafson、Matt Savage、Micah Martin、Michael Hale、Michelle Pace、Patrick Kua、Patrick Morrison、Ravi Mohan、Steven Baker、Steve Tooke、Tim Bacon、Paul Pagel、Enrique Comba Riepenhausen、Nuno Marques、Steve Smith、Daniel Sebban、Brian Brazil、Matthew Russell、Russ Miles，还有Raph Cohn，感谢他们与我通信，并把他们的思想和故事讲给我们，供我们使用。

2008年，我们启动了*http://apprenticeship.oreilly.com*，在上面公布了这些模式内容，以期获得来自社会的反馈。感谢所有为此作出贡献的人们，包括Julie Baumler、Bob Beany、Antony Marcano、Ken McNamara、Tom Novotny、Vikki Read、Michael Rolf、Joseph Taylor，特别要感谢Michael Hunger，他是论坛上的活跃分子，在多次手稿复审中为我们提供了极好的反馈。

我还要对联合太平洋捷运西线（Metra Union Pacific West Line）列车上每天的乘客们表示感激，这条线路从芝加哥洛普区（Chicago's Loop）开到西部郊区。本书的绝大部分内容都是在这列像图书馆一样安静的列车上写就的。感谢你们，在我撰写本书时，你们都静静地读着你们的书。明天见！

2006年，我加入了Obtiva公司，当时Kevin Taylor说服了我，他说我应该成为公司的第四名雇员，而不是转包商。这绝对是个好建议，而且我从许多方面得到了回报。我要感谢Kevin，感谢他支持我未经验证的想法，将公司的一部分交由我支配，并且收拾那些不断被我制造出来的麻烦，照应公司业务中那么多乏味透顶却又至关重要的方面。我为

公司的未来感到振奋。Kevin允许我实施的未经验证的想法之一就是建立Obtiva公司的软件工作室，来带领那些我们可以培养成高级开发者的软件学徒。自从2007年4月启动这个工作室，我们带出了六名学徒，我必须向前三名软件学徒（Brian Tatnall、Joseph Leddy和Nate Jackson）致以诚挚的谢意，他们忍受了我太多的缺点和缺乏经验的表现。这些小伙子们所经历的试验和错误使我们逐渐改善了最近三名学徒（Colin Harris、Leah Welty-Rieger和Turner King）的训练过程。感谢所有六位学徒，感谢你们的奉献、热情，以及常常在一个并不理想的环境中学习成长的意愿。

Mary Treseler就是那个鼓励我们发布这一项目的人。自从2005年她第一次读到我们最初的几个模式，就在心中产生了共鸣，虽然她自己并不是一名程序员。感谢你，Mary，虽然我们并没有写作的经验，你仍然愿意倾听我们的意图，而且这些年一直坚持耐心地同我们交流。

我很幸运，成长在一个非常和睦的家庭中。尽管我们搬了好几次家，但我的爸爸妈妈作为孩子的父母、彼此的爱人和基督信徒，处处给我们树立榜样。有了他们的行为榜样，我从成人、结婚到为人父母的过程都很顺利。从很早的时候，Marcia Hoover和Rick Hoover就成了鼓励我写作的持续动力。感谢你们，妈妈，爸爸，感谢你们开发了我喜欢写作的天性。

尽管我直到26岁才开始编程，但我并没有浪费组建家庭的时间，女儿出生时我24岁，在我研究生毕业的几个月前。尽管在那种情况下组建家庭颇具挑战性，但我的孩子们带给他们父亲的一样东西就是让我有了对自身责任的高度关注。自从1999年Rose出生以后，我不敢一天不工作，对一个正开启一条新的职业道路的人来说，这是极大的动力。随着孩子们从婴儿长大成学龄的孩子，我从观察他们克服学习过程中的障碍当中得到了启示。这提醒了我，让我谨记活到老学到老，像他们那样顽强地探求知识。Rose、Ricky，还有Charlie，感谢你们无条件地爱着老爸，并能忍受你们的第四个“兄弟”：老爸的笔记本电脑。现在这本书已经写完了，你们以后见到它的时间应该会少一些了。

我的妻子，Staci，嫁给了大学足球队的队长。11年之后，嫁给了一个带着黑色厚框眼镜的小伙子，这个家伙喜欢学习编程语言，并在闲暇时间发起新的编程用户组。这两个人都是我，Staci就这样跟着我一路走来，看着我同自己骨子里的那个奇客（geek）打交道。她忍受了我为所欲为地钻入了无休止的书籍、博客、开源项目、写作项目和雇主老板当中。没有人比Staci更了解我，也没有人比她更能拴住我的心。谢谢你，Staci Sampson Hoover，谢谢你使我能关注那些真正重要的事情。我会永远爱你。

最后，我要感谢上帝，感谢您无条件地爱着我，并在我很多次将要失落的时候把我救起。我希望我写的这本书能在某种意义上为您赞美。

Ade的致谢

首先我要感谢Dave感谢过的所有的人。没有他们，Dave就不会在现在的地方，我也不会。

我要感谢《The Pragmatic Programmers》（程序员修炼之道——从小工到专家）（Andy和Dave写的），是这本书给了我灵感，并把我带到了C2 wiki和极限编程周二俱乐部。如果没有这些事情对我的影响，我就不会找到Laurent Bossavit的Bookshelved wiki，也不会在Dave加入ThoughtWorks时知道他是谁。

当然，如果不是因为在一次老英国央行的XTC晚会上，Paul Hammant非要我对不想加入ThoughtWorks这件事给出合理的解释，我也不会成为那里的一名咨询师。谢谢你，Paul。在ThoughtWorks工作的经历为我打开了很多扇门。比如，有了ThoughtWorks前任创新总监（Innovation Director）Dave Farley的赞助，我能到Allerton参加PLoP会议，并见到Dave。

那些花时间为本书接受职业详细资料问卷调查的人们，你们肯定知道我说的是谁。我无法把你们的名字全部列在这里，但我永远感谢你们。对

于本书的审稿者也是一样。感谢你们付出的宝贵时间，告诉我们怎样才能使这本书更好。

Ravi Mohan不只跟我们分享了他的经验。针对本书以及软件技能的各个方面，他都向我们提出了一些很难的问题。他自愿地去做背景阅读，去改变自己的想法，并不断地询问问题的明确解释，这一切都让我们保持诚实。感谢你，Ravi。

我还要感谢Robert Konigsberg和Eve Andersson，感谢他们为本书手稿的早期版本提供了无比详尽的反馈。

我要感谢Enrique Comba Riepenhausen，感谢他们完成了最初的OmniGraffle图表（OmniGraffle是由Omni Group开发的基于Mac OS X系统的可视化制图工具）。若没有他们的帮助，你看到的将是一些相当丑陋的用Graphviz自动产生的图表（Graphviz是一套开源的可视化图形软件，来自AT&T研究中心）。

如果没有人指导，写一本关于学徒方面的书将是不可能的任务。Ivan Moore一直是我的指导者，虽然我们现在已不再在一起工作。Ivan的指导让我一直觉得快乐，就像我对茶的感觉一样。

我还要感谢Mary Treseler，虽然我们延误了多个“最终期限”，你依然给我和Dave完成这本书的机会。

最后，我要感谢我的父母。他们给我买了第一台电脑，而且他们在很多年之前就意识到我应该成为一名专业的程序员。如果我能在更小的时候就听他们的话，我的路将会更通畅更直接一些。

软件工艺宣言

注解

2009年3月，经过software_craftsmanship邮件列表中的长时间讨论，特起草如下宣言。

作为胸怀大志的“软件技师”，我们在亲身实践专业化软件开发，同时帮助他人掌握这一技艺，以此设立专业化软件开发的更高标准。基于这些工作，我们认为：

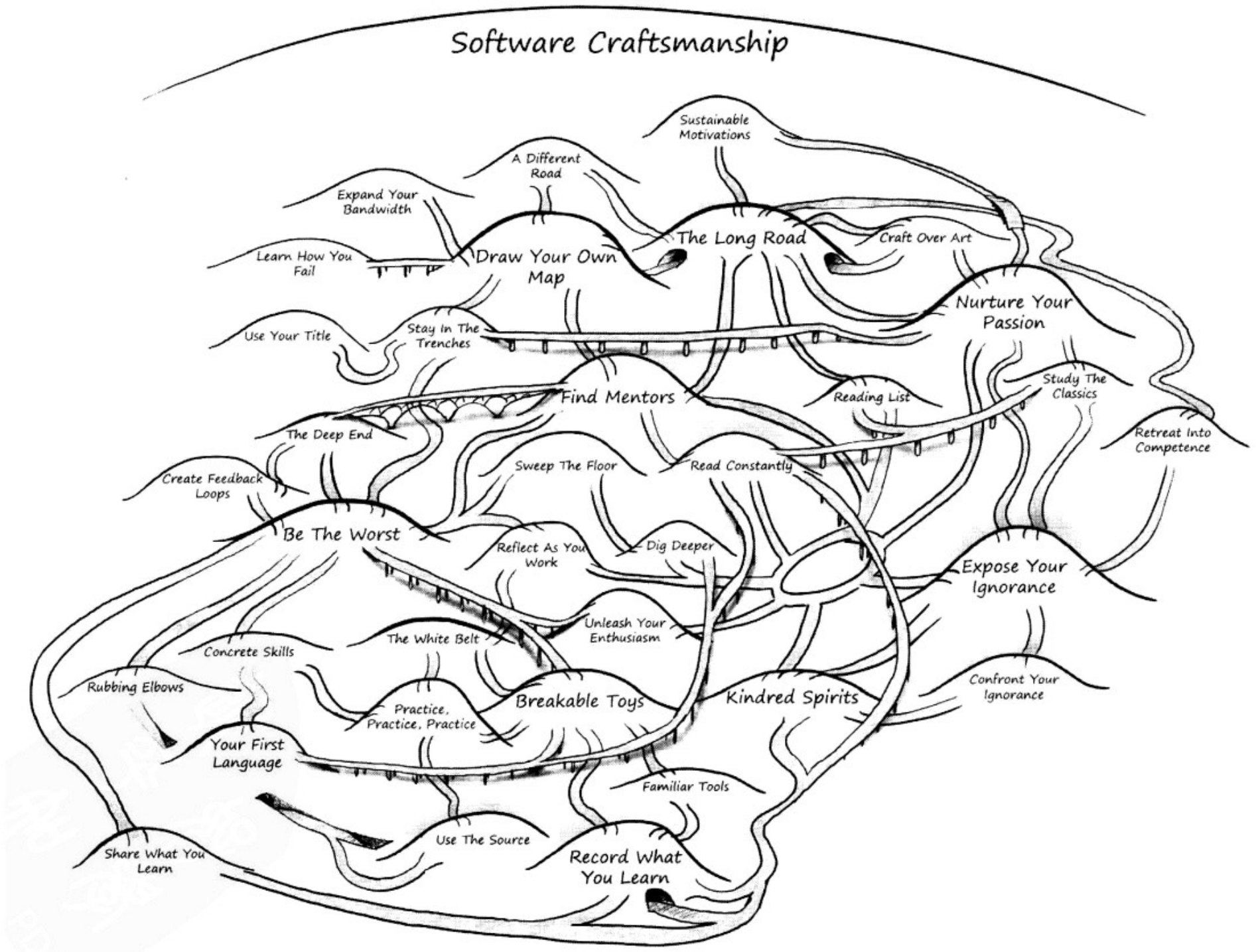
“可工作的软件”犹嫌不足，尚需精益求精的软件；

“响应变化”犹嫌不足，尚需稳步增加价值；

“个体与交互”犹嫌不足，尚需专家社区；

“客户协作”犹嫌不足，尚需卓有成效的伙伴关系。

也就是说，在追求左侧项目的过程中，我们发现右侧项目也是不可或缺的。


Software Craftsmanship
Sustainable Motivations
A Different Road
Expand Your Bandwidth
Learn How You Fail
Draw Your Own Map
The Long Road
Craft Over Art
Nurture Your Passion
Use Your Title
Stay In The Trenches
Find Mentors
Reading List
Study The Classics
The Deep End
Retreat Into Competence
Create Feedback Loops
Sweep The Floor
Read Constantly
Be The Worst
Reflect As You Work
Dig Deeper
Expose Your Ignorance
Unleash Your Enthusiasm
The White Belt
Concrete Skills
Rubbing Elbows
Practice, Practice, Practice
Breakable Toys
Kindred Spirits
Confront Your Ignorance
Your First Language
Familiar Tools
Use The Source
Share What You Learn
Record What You Learn

第1章

绪论

学徒期（apprenticeship）很重要，因为它能把掌握技能的
毕生热情灌输给你。它把永远学习的热情灌输给你，
并且在此过程中，使软件学徒变成了卓越的开发者。
——Pete McBreen，《Software Craftsmanship》（软件工艺）

本书是写给软件学徒（apprentice）的——写给对软件开发有一定体验并想继续深入，但同时又需要有人指导的那些人。无论你在享有声望的计算机科学专业拥有大学学历，还是完全靠自学的人，你看到世界上有一些优秀的开发者，而且渴望获得他们所拥有的精湛技能，那么本书就是为你而写的——不是为你的老板，不是为你的团队主管，也不是为你的教授。对于那些位置上的人，我们有很多其他的书可以推荐，但这一本是为刚刚启程的人准备的。

在写作本书的过程中，我们被《Software Craftsmanship》一书中的基本原则和思想深深影响。事实上，本书的书名也反映了这一点。学徒期的概念源于中世纪的手工艺模式，那时一小组从业者在一起工作，经验较少的学徒帮助学徒期满的熟练工（journeyman）和师傅（master）完成工作。我们写作本书的目的之一就是鼓励那些热爱创作软件的人专注于他们的技能。这里我们所讨论的职业过程是从“Hello World!”开始，但它何时结束呢？经常的情况是：一次向中层管理职位的晋升结束

了这一过程。太多有才能的人未经深思熟虑就接受了这种职位晋升，然后在短短的几年后发现他们根本不热爱自己的工作，并开始向往退休的日子。而对那些拥有熟练的软件开发技能并热爱学习的人们来说，软件开发将是可以持续一生的职业，而且会是一次风光无限的旅程。

在旅程开始之前，我们想告诉你Dave[译注1]的故事，并提供一些解说。他的故事展示了一种组合使用各种模式的方式，通过这些方式，不同的学徒模式结合起来产生强大的外力来促进软件开发者的个人成长。另一方面，这些解说是一种尝试，尝试将形成软件技能基础的一整套思想收集在一起，尝试回答跟这些思想有关的常见问题。

Dave的故事（讲述中包含了几个学徒模式的名字）

我的“入门语言”是Perl，但那也是之前两次学习编程失败之后的事。12岁时，在观看了电影Tron[译注2]之后，受到“一个完整的世界存在于我的电脑中”这种想法的激发，我尝试在自己的Apple IIe机器上自学BASIC。我买来一本由Apple公司出版的BASIC手册，但我想不出如何用这门语言做点让人觉得有趣的事情。当认识到除了基于文本的游戏，我根本写不出任何其他东西时，最终我放弃了。之后在25岁时，我阅读一本傻瓜书[译注3]来尝试自学Java，并慢慢地做各种练习，创建Java applet。结果我感觉自己真的成了个傻瓜，一切看起来都是那么难，我因此又放弃。直到26岁时，我找到了两位指导者，我的第一门语言才算学会了。那是在伊利诺伊州斯科基市，在“dot-com”泡沫的浪尖上，我在Edventions有

译注1：本书作者之一David的自称，英语中Dave是David的昵称，同样后面提到的Ade就是本书的另一位作者Adewale的昵称。

译注2：Tron，中文名“电子世界争霸战”，Steven Lisberger导演，1983年上映的一部电影。讲述黑客Kevin寻找安全系统程序Tron助自己毁掉主脑操控程序，从而让虚拟世界和现实时界都恢复秩序的故事。

译注3：傻瓜书（Dummies）系列是由John Wiley & Sons公司出版的系列书籍，所涉及的学科门类范围很广，如编程语言方面的《C for Dummies》、《C++ for Dummies》、《Java for Dummies》等。

限责任公司遇到了这两位指导者。公司的创始人Irv Shapiro知道我想成为一名程序员（他当时雇我做在线内容编辑）而且需要让我学习Perl，扑通一声，他把一本《Programming Perl》（*http://oreilly.com/catalog/9780596000271/*）（Perl语言编程）放在我的桌子上，并描述了一个“质脆玩具”作为我的练习作业。接下来的几天里，我啃完了《Programming Perl》，虽然对于像我这样的新手来说，这是本相当厚的书。为了继续探究Perl，我从自己的阅读列表中挑了一本消化起来更容易的可视化快速入门指导（Visual QuickStart Guide）系列的书。我的另一位指导者是Steve Bunes，Edventions公司的CTO，他时常坐下来与我“密切交往”，为我演示一些强大的调试技术，直到今天我还使用这些技术。在我完成第一版“质脆玩具”的过程中，最难运用的一个模式就是向旁边工作间中的那些有经验的Perl程序员和系统管理员“暴露无知”。但收起自己的面子是值得的，因为他们给了我一些快速的指导，使我弄清了程序中的一些问题，并理解了UNIX文件权限，这使我很快地完成了“质脆玩具”，让Irv和Steve大吃了一惊。

两年之后，我开始寻找一些机会，让自己的职业生涯超越我心爱的（但也越来越无法升值的）Perl并学习一些新技术。我一头扎进了极限编程（Extreme Programming，XP）和敏捷开发（Agile Development）中，让自己“提高带宽”，那时，极限编程和敏捷开发还处在宣传周期（Hype Cycle）的高潮中。我花了几天时间参加附近一所大学的XP/敏捷讨论会，从那里吸收新的信息。会见和聆听像Ron Jeffries[译注4]、Martin Fowler[译注5]、Bob Martin

---

译注4：Ron Jeffries，XP的3位奠基者之一，“敏捷宣言”最初的17位签署者之一，著有《Extreme Programming Installed》，《Extreme Programming Adventures in C#》等书。

译注5：Martin Fowler，企业级软件领域的著名作者、演讲人和咨询师，著有《Refactoring》，《UML Distilled》及《Analysis Patterns》等软件领域的名著，几乎本本都是畅销书。

“大叔”、Alistair Cockburn[译注6]和Kent Beck[译注7]面对这样的人，简直是一种令人陶醉的享受，我从讨论会中出来后便成了正式的面向对象和极限编程的超级崇拜者。我发现Joshua Kerievsky正在写作《Refactoring to Patterns》（重构与模式），这听起来真让人钦佩，于是我找了个“同道中人”一起学习。很快，我们发现我们都跑在自己前头了，因为我们甚至还不知道重构和模式是什么。于是我开始寻找更适合自己经验水平的书。最终找到了《Object-Oriented Software Construction》（影印版《面向对象软件构造》，机械工业出版社引进）和《A Pattern Language》（建筑模式语言）。我还是想以后再回过头来再看看《Refactoring to Patterns》，于是把它加到了我的阅读列表中。

还是跟那个“同道中人”一起，我从2002年开始学习Ruby，但我无法找到很多在日常工作中使用Ruby的途径，直到出现了Ruby on Rails。2005年，我再次拾起Ruby，努力寻找一些在日常工作中使用它的方法。我开始用它构建一个“质脆玩具”，却发现自己的思维方式太像Perl程序员。对于任何一个精通其“入门语言”的程序员来说，当他学习一门新的语言时，都会遇到求助于已有语言的标准和惯用法的诱惑。Ruby拥有优雅、简洁的美誉，而我所写的代码让人感觉既丑陋又蹩脚，这让我觉得自己是在做错误的事情。我刻意决定系上“白色腰带”，于是把自己的Perl经验丢在一边，开始深入钻研Ruby文档。之后，我很快明白了我需要的是什么，并将那些费解的代码重构成精良、标准的函数调用（Ruby程序员们好奇了？就是String#scan那样的）。为了让所有这些新知识留在我的大脑中，我决定“暴露无知”：在网站上记下了自己学到的东西，让所有的人看。[注1]

译注6：Alistair Cockburn，敏捷运动的发起者之一，著有《Agile Software Development》、《Writing Effective Use Case》等书。

译注7：极限编程和测试驱动开发的缔造者之一，“敏捷宣言”最早的17位签署者之一，著有《Extreme Programming Explained: Embrace Change》、《Test-Driven Development: By Example》等书。

注1：*http://redsquirrel.com/cgi-bin/dave/craftsmanship/ruby.white.belt.html.*

什么是软件技能

像工艺（craft）、技能（craftsmanship）、学徒（apprentice）、熟练工（journeyman）和师傅（master）这些简单的词汇，字典中的定义无法满足本书的需要。它们经常形成循环解释（“工艺”解释成“一个技师（craftsman）所拥有的技能”，而“师傅”被解释成“表现出技能水平的人”，而“技能”则被解释成“将手工艺传统中的技师连结在一起的一种品质”），很少基于特定国家行会体系的历史，而且常常被泛化到用于描述通过某种技能构建起来的任何东西。简而言之，这些定义没有排除任何东西，于是也就包含了所有东西。而我们需要的不止这些。

关于“软件技能”（Software Craftsmanship），网络搜索引擎列出了61 800条引用，但对一个寻求职业指导的人来说，这些引用很少有提供有用解释的。让人郁闷的是，许多这样的文章都是由一些善意的程序员写出，他们发现在这些混乱的相关概念中隐藏着一些有用的东西，却没有能力把这些东西明确地提取出来。

Pete McBreen的《Software Craftsmanship》是一次尝试，尝试为软件开发的新颖方法整理出一份宣言，一般认为“软件开发是一门工程学科或一门科学”，而McBreen的书所瞄准的是那些做事并不基于此种假设的人。但这份鼓舞人心的工作仍有瑕疵。他没有把今天正在实践着的软件技能跟他希望实践的软件技能区分清楚。他也没有把他的愿景（vision）跟“隐形的行会监控高度成熟的行业”这种中世纪的手工艺观念做一个足够清楚的区分。他犯了一个从软件工程的反方向来定义软件技能，并要求读者在二者中做出选择的错误。而我们认为，一种工艺模型可以用更加积极的方式来定义，人们觉得构建软件工程学科还是有意义的，我们的定义不能把这些人排除在外。

我们从中世纪直到工业革命前盛行于欧洲的手工艺制度（《The Craftsman》（工匠），第52～80页）中获得启示。在那种制度中，行会控制着师傅，师傅控制着那些工作和生活在工场中的人。师傅拥有工场并有绝对的权威。在这种严格的等级制度中，位于他们下方的是熟练

工。他们通常都是能工巧匠，但仍需完成他们的“杰作”，以此表明他们的技艺已足够精湛，可以做师傅了。

技工们四处游走，正是借助这一点，新技术才从一个城市传到另一个城市。除了引入新技术，技工们还要监管学徒们的日常工作。学徒们会为一位师傅工作多年，直到他们能证明自己已经掌握了基本技术，并理解了手工艺的价值，从而升级为一名熟练工。不能被安放到行会等级结构中的人是不能合法从业的。

可以想象，这一系统在今天即使没有违法，也肯定要遭到唾骂，而且根本不切实际。我们不想再重复这种将手工艺制度推到了现代社会的边缘的错误。相反，我们相信完全可以放弃匠人工场的传奇式幻想，而采用现代工艺工作室，在那里我们可以在过去的基础上不断改进，而不仅仅是模仿它。

我们从敏捷开发运动中学到的经验之一是：仅仅告诉人们去做事情并不能带来长久和可持续的变化。按照你的要求去做事的人如果遇到你的规则没有囊括的新情况，他们马上就会迷失。相反，还是同样的人，如果理解了支撑规则的底层因素，他们就能想出新的规则来适应任何情况。在这里，我们的目标不是简单地交给人们一本讲解规则的书，而是给予他们针对新情况创造新实践，进而将软件开发学科推向前进的能力。

我们对软件技能的愿景，一部分是价值的提取，针对我们为写作本书所访问的技艺精湛的个人；另一部分是希望的表达，表达我们希望出现的社区类型。书中的思想是这一愿景的起点。因此，当我们使用短语“软件技能”，我们所说的是由一些相互交叠的价值所联结和定义的一整套实践，包括：

对Carol Dweck所做的研究，即提倡“成长型思维模式”（Growth Mindset）的依属。这带来了一种信念：如果你愿意钻研一件事，你就能做得更好，一切也将得以改善。用她的话说，“努力是使得你聪明能干的东西”（《Mindset》（心理定向与成功），第16页），而失败不过是引导你下次尝试不同方法的激励机制。这跟

以下信念是相反的：我们每个人天生都带着固定数量的禀赋，而失败就是我们天资不够的证明。

- 基于你从周围世界获得的反馈，始终不断适应并做出改变的要求。Atul Gawande将这说成是一种“从你所做的事情中寻找不足之处并寻求解决办法”的意愿（《Better》（更好），第257页）。
- 一种对注重实效而非教条主义的向往。这包括一种肯于牺牲理论上的纯洁性和未来的完美而达到“今天把事情做完”的意愿。
- 一种认为分享知识胜过隐藏独享的信念。这常常关联到对自由和开源软件社区的参与。
- 一种敢于实验并被证明错误的意愿。这意味着我们可以什么都试试。我们失败了。然后我们在下次实验中运用这些来自失败的经验。正如Virginia Postrel所说：“并非每种实验和想法都是好的，但只有尝试新的想法我们才能获得真正的进步。要做的事情总是可以更多。每一次进步都可以被继续改善；每一种新的想法也会使更多的新组合成为可能”（《Future Enemies》（未来的敌人），第59页）。
- 一种心理学家称之为“内控倾向”（internal locus of control）的精神。[注2]这包括掌控自己的命运并为之负责，而不是等待别人给我们答案。
- 一种对于个体而不是群体的关注。这不是一场“领导者－追随者”式的运动。相反，我们是一群想要提高自身技能的人，我们发现：争论、反对、分歧（而不是盲从于自诩的权威）正是到达目标的途径。我们相信我们都处在同样的旅途中，我们追寻的是自身的改变，而非世界的改变。这也解释了为什么本书不关注团队的调整，而关注提高自身技能的途径。

注2：*http://en.wikipedia.org/wiki/Locus_of_control.*

- 一种包容性。我们不抵制企业级软件开发、计算机科学或软件工程（事实上，Ade目前的职位头衔中就含有“工程师”一词）。相反，我们认为，一种有用的系统应该能够从软件开发社区的所有元素中识别并吸收那些最好的思想。
- 我们以技能为中心，而非以过程为中心。对我们来说，拥有高超的技能要比使用“正确”的过程更重要。基于这种思想可以得到一些推论。Gawande问，“医学是一种工艺还是一种行业呢？如果医学是一种工艺，那么你应该关注如何教产科医师掌握一套工艺技能……你进行研究以发现新的技术。你承认并非每个人都能把事情搞定”（《Better》（更好），第192页）。这种思想隐含这样的意思：没有哪种过程或工具能使每个人都同样成功。尽管我们都能提高自己，但我们的技能水平总会有差异。
- 对Etienne Wenger所谓“情景学习”（situated learning）的一种强烈偏好。[注3]这种思想是说，软件社区应该尝试抓住像“听力范围内的专家”（Expert in Earshot）这样的模式。[注4]其要旨是：最好的学习方法，就是同那些使用你要学习的技能来达到某种目标的人处于同一个房间里。

这种价值体系针对不同的责任引出了不同的角色，正如后续的部分所讨论的。

做软件学徒意味着什么

谈到做学徒意味着什么，我们的访谈者之一Marten Gustafson说得最好：“我猜它基本上是指拥有这样一种态度：对于已经做完或者正在做着的事情，永远都有一种更好、更聪明或更快的方法来完成它。而学徒期就是这样一种状态或过程：不断演进并寻找更好的方法，找到能使自己学会那些更好、更聪明或更快方法的人、公司和情景。”我们认为，拥有这种“不依赖于任何人向你提供方案，靠自己找到处理问题的建设

注3：*http://c2.com/cgi/wiki?LegitimatePeripheralParticipation.*

注4：*http://c2.com/cgi/wiki?ExpertInEarshot.*

性方法”的内在动力是非常有价值的。Dweck曾经写过：“它不是一种靠成功来增加并随失败而减少的内在数量……它不是一种我们只需告诉别人他有很高的智商就能给予那个人的东西。它是一种我们为别人提供装备然后让他们自己去获取的东西——方法是教他们重视学习而不是外在的聪明，教他们学会享受挑战，并将错误看作臻于精熟的通道。”（《Self-theories》（自我理论），第4页）

虽然最理想的情况是把你放在一个拥有学徒伙伴、熟练工和师傅的小团队中，我们对于学徒期的理解并不要求这样的安排。你的学徒过程由你自己掌控，而最终的训练成果就是你的责任感。尽管学徒期的进展过程都由你自己来决定，但是否有人指导，以及指导人的品质也会对你的技能产生持久的影响。

作为一名软件技师，学徒期是你旅程的开始。在这段时间里，你将首先内向地关注自身，下决心提高自己的技能。虽然你会受益于同伴的关注和经验更丰富的开发者的关注，但你必须学会自我成长，学会如何学习。做学徒意味着什么？本质就是这种对自身的关注和提高自身技能的要求。

最终，学徒将从一个除了持续学习很少有其他责任的位置成长到一个拥有更多外向型责任的位置，我们一般会相信：这种转变只有在回首往事的时候才能看出来。在某个时间点，学徒得到了师傅或熟练工的评价，并被告知他在社区中的工作和角色已经处在熟练工的位置上了。在这种情况下，其实学徒早已经开始担当更多的责任了，像一只“被水煮的青蛙”，他经历了一种渐进而非离散的转变，从一种状态转到了另一种状态。一些人完成这种转变需要的时间会比别人更长。对某些人来说，这种转变所花的时间甚至超过了整个职业生涯。

做熟练工意味着什么

在你的技能水平不断进步到新的阶段时，你会保留前一阶段的特征。因此，跟学徒一样，为了在技艺方面不断地进步，熟练工和师傅将保持一种内向的对于自身的关注。同时，另一种新的关注会出现在熟练工身

上，那就是对从业者之间关系的关注，以及对团队内外的沟通渠道的关注。以前，熟练工会从一位师傅转到另一位师傅那里，并沿途在不同的团队中间散布思想。现代软件开发的现实是：你会在相当长的时间内与单个团队呆在一起，这时，你会关注如何改善团队内部人与人之间的相互关联。这种关注最终会扩展成一种为身边的人提供指导，并同其他业内人士保持沟通的责任。

熟练工关注如何构建一些能彰显其技艺进步的更大的应用程序；他在不同的项目和师傅之间移来移去，力求拓展其技艺组合的广度和深度；他力求提升自己在社区中的位置；并努力为成为师傅做好准备。

熟练工的责任比学徒更广。同样，他的失败也会带来更多的损失。我们将要讨论的模式中有一些并不适于熟练工，这完全是因为对那些视之为指导者的人来说，他负有更大的责任。

做师傅意味着什么

精通意味着行使学徒或熟练工的所有职能，同时还要关注如何将行业向前推进。“达到技能和技术的炉火纯青”（《The Creative Habit》（创造性的习惯），第167页）只是一个开端。精通还包括掌握一种技能，然后把它变成一个能将其他人的技能提升几个数量级的放大镜。这会表现为创造直击软件开发本质的新工具，也可能是训练一代熟练工，使之青出于蓝而胜于蓝。或者，它会表现为我们还没想到的某种形式。总之，师傅将高级技能的获取、使用和分享看成是身为软件技师的重要方面。

这里所讨论的学徒、熟练工和师傅的定义跟你在任何字典上能找到的都不太一样。这是一些新的解释。但我们相信，我们的软件技能愿景中存在固有的价值，它能够帮助你，使你希望自己有多成功便能有多成功，不管你最终采纳了它们，提高了它们，拒绝了它们，还是遵循了一条完全不同的路。

学徒期是什么

最基本的学习情形是这样：一个人帮助一个知道自己
正在做什么的人，从而让他学到东西。
——Christopher Alexander等，《A Pattern Language》，第413页

许多书，包括1945年版的《Fifteen Craftsmen On Their Crafts》（十五工匠学艺，第69页），都描绘了一种学徒训练的老套情景：一个十来岁的小男孩，脸上满是烟灰，在一个铁匠铺子里做工。铁匠，那个粗鲁的、有经验的工匠，在男孩的协助下打造着自己的项目。有时男孩积极地参与到这一过程中；也有时他只是在打扫店铺，但仍然密切注意着正在工作的师傅。最典型的情况是：男孩的学徒期将持续好几年，除了知识、经验和食宿，他很少能得到其他报酬。

最终，男孩将学到足够的技艺来独立承担一个项目，甚至可能离开他的第一位师傅，到另一家店铺中充当责任更多的角色。学徒期结束时，他将获得一份铁匠的营生，靠自己的手艺来为自己遮风雨、充饥肠，并购置工具。在现今世界上，包含一名熟练的软件开发者和一名新手的学徒训练跟这种旧式的学徒训练已经少有相似之处了。那么，我们现在对学徒期的理解是怎样的呢？它又如何超越那种老套的情景呢？

明确一点，在本书中我们并不是要向软件开发新手建议理想的做学徒的方法。如果我们是为团队负责人和项目经理写这本书，那么对于构建理想的学徒期提供一些指导是有意义的，因为这些人确实有力量推动这类训练过程。但本书是为软件开发的新人而写的，这些人处在阵地上，想努力搞清楚怎样才能学到他们需要的东西，从而实现一些目标，比如找一份（更好的）工作、完成项目，或者成为一名卓越的开发者。由于大多数新人的经历并不与一种“理想的”学徒期相似，现代学徒期的概念主要是一种心境：你认识到自己处于起始阶段，哪怕你已编程多年；而且你想采取措施从你所处的环境中建立自己的学徒过程。

大多数人都没有机会参与一次由软件熟练工指导的正式学徒训练。现实中，多数人都只能在不太理想的环境中磨砺他们的学徒过程。他们

可能面对傲慢专横或能力不足的经理，精神涣散的同事，无法达成的最终期限，以及条件低劣的工作环境：把新手当苦力，安置在狭窄的长方形隔间里，有一台PC跟一条很卡的网络连接。本书中所有的经验都来自于那些为达到新的技能水平而不得不克服这类低劣环境的人（就像我们）。在这个行业注意到后面所讲的Pete McBreen的建议之前，新手仍将需要像本书的这样书籍来帮他们创造自己的学习机会。

> 我们可以从容不迫地培养学徒开发者，因为我们面对的是过剩的问题，而不是短缺的问题……如今的开发者数量比我们需要的多，而我们缺少的是好的开发者。
>
> ——Pete McBreen，《Software Craftsmanship》，第93页

学徒过程是学习如何成为专业软件开发者的途径。特别地，它是学习如何成为你所找到的技能最高超的软件开发者的途径。它包括寻找好的老师并利用在他们身边工作的机会来学习。它是成为一种不同的软件职业者的道路上迈出的第一步，这种职业者想要的绝不只是“称职”。

学徒模式是什么

学徒模式（Apprenticeship Pattern）尝试给那些走在职业进步的道路上按工艺模式来工作的人提供指导。所有的模式都是从我们自己和那些我们访谈过的人的经验中提取而来。跟任何好的模式一样，它们会引起你的共鸣，让你觉得这并非玄门独创，因为你周围的人已经在使用它们。这些模式的另一共性是它们的生成性（generativity）。每次你运用它们都会得到不同的结果，如果用在合适的上下文中，它们会改善你的工作环境。它们并不是每次运行都保证得出同样结果的算法。相反，它们是解决一组问题并造出新问题的工具。诀窍在于：你要用自己的判断来选择自己更想解决的问题。

本书是以模式语言的形式来组织的。模式语言是针对特定领域中常见问题的一组相互关联的解决方法。最初的模式语言是Christopher Alexander在《A Pattern Language》一书中所写的，在那本书中，他描述了250多种用于设计各类建筑的模式，从厨房到房屋到城市甚至

社会。Ward Cunningham[译注8]和Kent Beck在1990年把模式语言引入软件行业，结果引来了大量的文献、书籍甚至聚焦设计模式的学术会议。软件设计模式著作中最有名的例子是“四人帮”写的《Design Patterns》[译注9]，而Martin Fowler的《Refactoring: Improving the Design of Existing Code》，（重构：改善既有代码的设计）是模式语言方面更好的例子。明确地说，你正在读的这本书并非一本关于如何设计软件的书，而是一本关于如何设计软件开发职业生涯的开端并为你在该领域成就卓越打下基础的书。

模式来自哪里

好的软件框架设计考虑的原则之一就是从能够运行的实际系统中提取框架元素。同样，软件设计模式就是从许多运行系统中提取出来的，那些系统中使用同样的方法来解决类似的问题。本书的内容最初提取自Dave在学徒期中经历的故事，之后基于Ade的故事做了检验和增补，最后用大约30位从业者的经验做了检验，这些人的开发经验从两三年到几十年不等。我们与这些人做访谈，目的是检验这些模式是否真是针对常见问题的一般解决方法，顺便也挖掘我们没有认识到的其他模式。我们还参加了多种研讨会（2005年的PLoP）、亚特兰大敏捷会议，以及ThoughtWorks的内部会议，以此来帮我们改善这些学徒模式的结构和精确性。最后，我们把这些材料的大部分都在网上免费发布，从社区中寻求反馈。

下一步做什么

随着你开始学习模式本身，要记住：你可以基于任意的方式选择、合并

译注8：Ward Cunningham，美国程序员，世界上第一个wiki的开发者，设计模式和极限编程方面的先锋。

译注9：全称《Design Patterns:Elements of Reusable Object-Oriented Software》，中译本《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，机工出版社还出版了该书的同名影印版和双语版。该书的作者一共有四位：Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides，常被称为“四人帮”。

或调整它们来适应你的独特情形。要理解：这些模式是针对特定上下文中的特定受众而写的。今后，有些模式会突然变得适合于你，而后，同样又突然间让你觉得不再合适。在你的职业生涯中，学徒期是你关注自身成长超过所有其他东西的一段时间。这是你暂缓直接最大化收入潜力的野心，从而最大化学习机会的一段时间。正因为此，它是一段你可以适当自私自利的时间。一旦这段时间结束，你的优先级将需要调整。你将不再是一名学徒，尽管你仍有很多东西需要学习；而你的优先级需要转向其他人：你的客户，你的同事，还有你所在的社区。

第2章

空杯心态

你没看到杯子已满茶水正往外溢吗？

——年轻的哲学家

一位年轻的哲学家经过长途跋涉去访问一位禅宗大师。大师同意与他见面，因为哲学家随身带来老师们写着高度评价的推荐函。两人坐在一棵树下交谈，话题很快转到大师可以向年轻人传授什么这个问题上来。大师感受到了年轻人的激情，他温和地微笑着，开始讲述自己的禅定术（meditation technique）。哲学家打断了他，说："嗯，我明白你在说什么！我们在寺院中使用过与之类似的技巧，不同的是我们使用塑像来定神。"

当哲学家向大师解释完自己是如何学习并实践禅定的，大师接着往下说了。这一次他试着向年轻人解释人应怎样与大自然和宇宙保持和谐。还没等他说完两句话，年轻人又一次打断了他，并开始谈论自己如何学习禅定术，以及很多很多其他的事情。

大师再一次耐心地等待年轻的哲学家结束他动情的解说。当哲学家再一次安静下来之后，大师开始讲述如何通过各种情形看出几分诙谐。年轻人见缝插针，又开始讲述他最喜欢的笑话，还有他认为那些笑话可以怎样联系到自己所经历过的场景。

哲学家说完之后，禅宗大师邀他到室内做一次茶道。哲学家听说过这位大师的茶道技艺与众不同，于是高兴地接受了。跟这样一个人品茶论道，这永远都是特别的礼遇。一直到进入室内，大师都做得完美无瑕，直到他开始往茶杯里倒茶。这时候，哲学家注意到茶倒得比平常满。大师继续往茶杯里倒，茶很快满到杯的边沿了。年轻人不知道该说什么，开始诧异地看着大师。大师仍然继续倒，就像什么也没发生一样，茶开始溢出了，热茶溅到地毯和大师的衣服上。哲学家不敢相信他的眼睛，终于忍不住大叫："不要倒了！你没看到杯子已满，茶水正往外溢吗？"

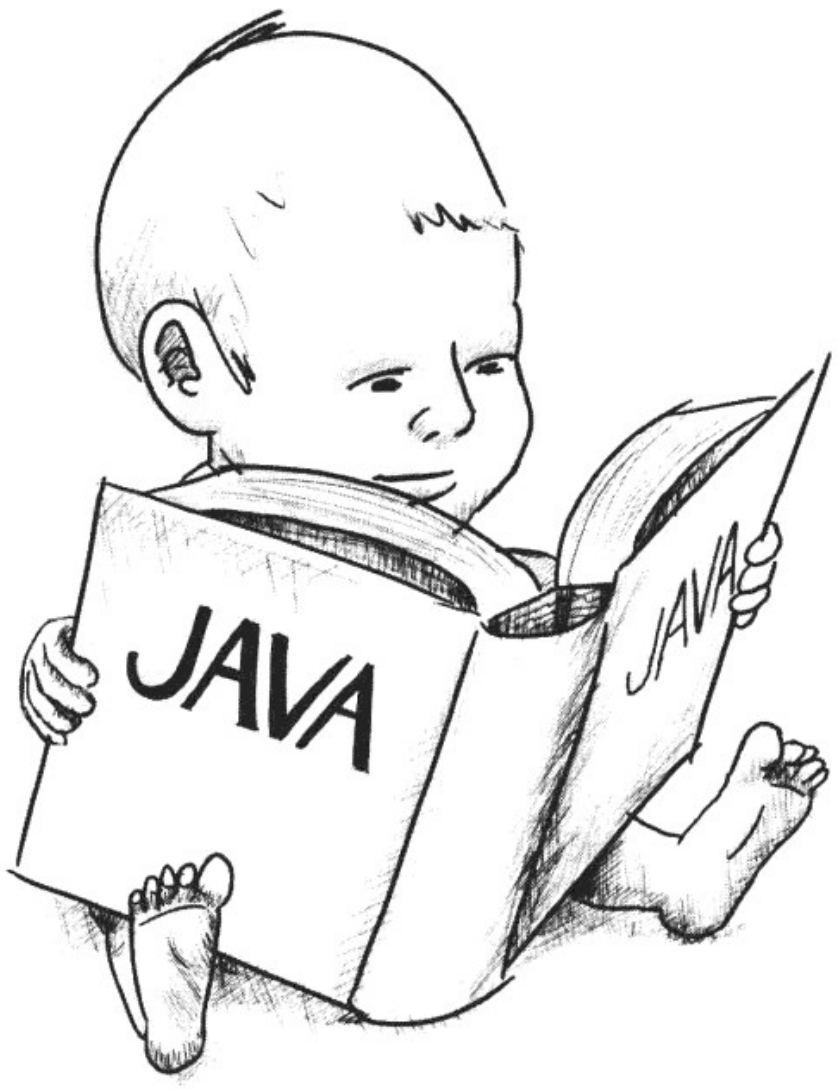
听到这句话，大师轻轻把茶壶放回火炉上，用他那恒久温和的微笑看着年轻的哲学家，说："如果你带着一只已满的杯子来找我，如何能指望我再给你一些东西喝呢？"

这个故事是从Michel Grandmont的“品尝一杯新茶”（Tasting a New Cup of Tea）[注1]改编而来的。我们在这里重提这个故事是为了阐明成功的学徒所需的一种态度。已有的经验越多，你就越需要更多的努力进入到“空杯”状态，清除思想中的坏习惯，放下对技能水平的自鸣得意，敞开自己，从更有经验的同行那里学习不同的而且常常是违反直觉的新方法。

本章的模式将提供一些方法，使你能顺利开启学徒期的旅程，并保持开放的思想。系上“白色腰带”表示不管你原来有多少专长，都要维持一种初学者的思想。“释放激情”将推动你跨越诸如灰心、迷茫和沮丧之类的新手障碍，使你敢于深入探究自己的“入门语言”。特定技术领域的“具体技能”会向你敞开大门，为你提供探索后续章节中更高级模式的机会。但不要让自己变得过于舒适！将最后四种模式合在一起使用，从而系统地获取知识，不断拓宽自己的技能领域。让自己在特定技术领域“暴露无知”，从而使注意力集中到下一步要学习的东西上面。然后“正视无知”，让团队和客户看着你雄心勃勃地获取知识。最终你将有机会承担一项冒险的任务，一次跳入“深水区域”的机会，要么学会游泳，要么沉入水底。这听起来挺吓人，但在你的职业生涯中没有比这更好的时机来尝试这类冒险了。当所有这些新知识和新思想的拓展快要把你淹没的时候，“以退为进”很重要，回想一下自己走了多远，开发了多少技能，然后打起精神去占领下一个高地。

注1：*http://www.ironmag.com/archive/ironmag/2000_mg_reality_of_learning.htm.*

入门语言



让大脑放下所有不重要的东西，一种好的记号（notation）
能使大脑关注于高级问题，效果上就相当于提高了
人类的智能水平……任何专业和行业的技术名词，
在那些没经过专门训练的人们看来都是难以理解的。
但这并不是因为它们本身很难。相反，
它们无一例外地是为了让事情更简单才被引入的。
——Alfred North Whitehead，《An Introduction to Mathematics》（数学引论）

情景分析

你刚刚起步，只对一两门编程语言有肤浅的认识。

问题描述

你感觉自己的职位取决于你是否能交付一套用特定语言编写的解决方案，而且质量上达到队友同样的水准。或者，你想获得一份工作，而这首先取决于你对某一门编程语言的精通程度。

解决方法

选择一门语言。熟练地使用它。接下来的几年里，这将成为你解决问题的主要语言，而且只要你还在实践，这都是你要磨炼的默认技能。做这个决定是一次挑战。将各种选项仔细权衡一下非常重要，因为这是你构建早期职业生涯的基础。

如果别人要你去解决一个问题，而这项任务指定要用某种编程语言，那就让解决问题的动力来指导自己的学习。如果你想获得一份工作，而这份工作要求你使用某种特定的编程语言，那就使用那种语言来构建一个玩具应用，最好是个开源项目，这样你所期望的雇主就很容易看到你写的代码示例。不论哪一种，去找一名你知道的、接触得到的而且是最有经验的程序员，以后需要帮助时就找他。解决一个问题是花费几分钟还是几天，就看有没有一个可以随时帮助你的人。但还要记住，不能依赖那个更有经验的朋友来解决你所有的问题。

在学习第一门语言的过程中，一种改善学习体验的基本方法就是找一个实际问题来解决。这可以使你的学习根植于现实世界，从而为你提供第一个较大的反馈回路（feedback loop）。基于书本或者文章中那些短小的、人为设计的例子来学习是有局限性的，你将失去学以致用带来的好处，毕竟，你在工作中要做的是解决问题。改善这一体验的基本方法是寻求反馈回路。特别地，创建较短的反馈回路能协助测量你的进步。有些语言相比其他语言拥有更好的反馈工具，但不论哪种语言，你都能采取一些措施来搭建一个学习沙箱（sandbox），然后在里面实验。

Ruby中有一个交互式的命令行工具：irb。Rails中有script/console。类似地，Erlang有erb。Firebug提供了很多好用的方法，可以在Firefox网

页浏览器中研究运行时的JavaScript。许多语言都提供了与此对等的工具。

有时，这些工具还不够，你需要更大的沙箱。Dave喜欢在他的IDE中始终放置一个空的Java类，当需要研究一个不熟悉的API（Application Programming Interface，应用编程接口）或者一项新的语言特性时，就可以直接拿来用。

```
public class Main {
    public static void main(String[] args) throws Exception {
          System.out.println(/* 这里要将要把玩的东西 */);
    }
}
```

当学到足以编写实际代码的时候，测试驱动开发（test-driven development）技术能让你基于小步骤前进，并确保自己的假设被检验。测试驱动开发是如此普及，当发现一门语言没有测试框架时，你甚至会有手足无措的感觉。不要犹豫是否写一些简单的测试来检查你对语言的理解，要么就直接让自己熟悉测试框架。

从采取近乎愚蠢的小步骤开始；等到你学会更多的东西，步子可以相应地放大一些。例如，Ruby语言有一项特性，可以把一个代码块应用到一个列表（list）中的每个元素上，并将结果收集到新的列表中。你可以编写如下的代码来明确你对这项特性的理解：

```
require "test/unit"

class LearningTest < Test::Unit::TestCase
   def test_my_understanding_of_blocks_and_procs
       original = [1, 2, 3]
       expected = [2, 3, 4]
       p = Proc.new { |n| n + 1 }
       assert_equal expected, original.map(&p)
   end
end
```

学习测试不一定只为了学习语言；也可以把这一过程用于学习其他人的库如何工作。久而久之，这些开发商的测试（Ade在伦敦测试自动化会

议[London Test Automation Conference]的一次闪电演讲中使用了这个名字），[注2]可用来检验当一个库升级到新版本时会不会弄坏你的系统。如果弄坏了，这些测试便可指明新库就是问题的源头，因为这些测试使用的功能恰恰是那个库提供的。在一个构造良好的系统中，可以用它们来验证一套库的不同实现是否拥有了你所需要的所有功能。

最后，你将不再编写用于学习的测试，而是编写用于检查实际代码的测试，这些测试不再检查你对语言结构和API是否理解。时间长了，你会发现除了简单的单元测试（unit test）外，还有很多其他技术可用于验证你的工作，以及你跟其他团队成员的交流。

下面的讨论是关于如何通过学习一门新语言来学习不同的思考方式，但Ralph Johnson（《Design Patterns》，中译本《设计模式：可复用面向对象软件的基础》［机械工业出版社］的合著者之一）的建议也适用于第一门语言的学习。

问：那假如某个人确实想学习怎样以不同的方式来思考，他该学习什么语言？Ruby、Python或者Smalltalk？

答：我倾向于Smalltalk。但我倾向于什么并不重要。你应该根据周围的人来选择一门语言。你知道周围有谁爱好这些语言中的某一种吗？你可以常与此人交谈吗？或者更好一点，你能跟此人一起做个项目吗？

就我所知，学习语言的最好方法就是和一位该语言的专家一起工作。你应该根据自己认识的人来挑选一门语言。一位专家就够了，但一定要有一位。

最佳的情形是：基于一个项目，使用那种语言的项目，你可以经常性地与专家一起工作，哪怕只是每周四晚上一次也行。另一种情形也一样好：你独立完成一个项目，但每隔两周你都在午饭时把代码样本带给专家看。

注2：Ade Oshineye，“Testing Heresies”（测试邪说），参见：*http://www.youtube.com/watch?v=47nuBTRB51c#t=23m34s*。

> 靠自己也可以学习一门语言，但除非与专家交流，否则你需要更长的时间来领悟语言的精神。
>
> ——Ralph Johnson谈语言学习[注3]

Ralph的建议与“找人指导”模式紧密相关，而且直接阐明了指导者对你的学习所能产生的影响。因此，在选择第一门语言时，是否能从就近的语言专家那里得到反馈就成了一个首要的考量。还有一点：选择了一门语言，就意味着选择了一个虚拟的实践社区，这个社区里有成型的惯用思想、社交集会方式和沟通机制。你应该利用这种支持网络，这样你就不只是在学习一门语言，而且实际加入了你的第一个“同道中人”社区。开始时，你所拥有的一切就是这个社区所倾向的工作类型，社区的边界，还有社区中的偏见和信仰。当选择学习一门语言时，你应该参加本地的语言狂热者集会（或者访问一个他们经常去的网上论坛），看看自己是否愿意属于这个社区。

如果能加入一个肯共享代码的社区，带来好处之一就是可以超越对简单语言结构的学习，并开始用惯用的语言思想来表达自己。但这只是开始，每一种语言都有其精妙之处，仅通过阅读别人的代码很难领悟到。

比如，XSLT中有“Muenchian方法”，Perl中有“Schwartzian转换”，而C中有“Duff设备”。这些技术都可以通过阅读代码学到。但是要领悟到它们为何重要，以及何时使用它们，则需要社区分享的经验。有时这种共享池仅存在于口头的传统中，你必须单独去跟一个人交谈才能获得这种知识。有时这种知识仅存在于邮件列表的归档中，或者一本在线的入门手册中，若没有上下文，很难理解它的重要性。在这类情形中，学习语言的新手不得不多年沉浸在社区中，才能把知识的管子接入到这个共享池。但现在这些语言的精妙之处常常被总结在一些书中，像《Effective Perl Programming》（Addison-Wesley出版）、《Effective Java》（Addison-Wesley出版）（中译本由机械工业出版社

注3：*http://groups.yahoo.com/group/domaindrivendesign/message/2145.*

出版）、《Effective C++》（Addison-Wesley出版）。在掌握基本语法之后马上阅读这类书籍能大大加速你的学习过程，并帮你避免常见错误。

所有这些都能帮你深入挖掘第一门语言。几年之内，第一门语言将是你学习其他语言的框架。第一门语言学得越好，下一门语言学起来就越容易。尽管你将主要使用这门语言来解决日常问题并交付软件功能，你还是应该经常花一点时间来延伸自己在正常工作中对它的使用。在非常规的方向上延伸一下语言的使用能帮你发现语言的强项和弱点。

Eric Merritt的网志“The Shape of Your Mind”（你思想的形态）深入讨论了编程语言会对你解决问题的能力产生多么深远的影响。

> 从很多方面来说，编程语言都像一种塑形工具，像那些给帕拉卡斯[译注1]婴儿塑头盖骨、给汉族女人缠足或者给克伦邦[译注2]长颈族女人塑颈的东西。在编程语言的情形中，被塑的不是头盖骨，而是我们思考问题、形成想法并把这些想法运用于特定问题的方式。比如说，如果你写只用早期版本的Fortran（Fortran 77或更早）写过代码，很可能你不知道递归的存在。同样，如果你只用过Haskell，你对命令式（imperative）风格的循环也会知之甚少。
>
> ——Eric Merritt，“The Shape of Your Mind”[注4]

深入挖掘入门语言也有危险，那就是深陷其中而止步不前。这门语言将在你整个职业生涯中成为你的“母语”而一直陪伴着你。不要让你对它的精通阻碍了自己对其他语言的学习和使用。健康的职业生涯会把你带入软件开发领域多姿多彩的语言洞天。每一种语言都为你提供了使用不同模式来解决问题的机会。在逐渐超越第一门语言的过程中，你应该寻找机会去学习一些采用迥然不同的方法来解决问题的语言。惬意于面向对象语言的学徒应该探究一下函数式（functional）编程语言。畅然于

译注1：Paracas，南美洲秘鲁地名，隶属秘鲁西南伊卡地区皮斯科省。

译注2：Karen，缅甸邦名，位于缅甸东南部。

注4：*http://erlangish.blogspot.com/2007/05/shape-of-your-mind.html*。

动态类型的学徒应该钻研一下静态类型。安逸于服务器端编程的学徒应该考查一下用户界面设计。一路走来，你肯定会对某些语言和解决问题的方法产生偏好，但要避免固执的、宗教式的亚文化把你推向一种“一刀切”（one-size-fits-all）的方法。这种语言技能的拓展是向着“师傅”般的博学多艺迈出的第一小步。

> 你不应该“嫁”给任何特定技术，而应该有足够宽的技术背影和经验基础，使自己能针对特定的情景选择好的解决方案。
>
> ——Dave Thomas和Andy Hunt，《The Pragmatic Programmer》（程序员修炼之道——从小工到专家），xviii页

行动指南

找一份语言规范（specification）来读一读。对某些语言来说，这很简单，去弄一本公开发售的书来看看就可以了。对另一些语言，可能只有一份语法规格给你读。甚至还有一些语言，其规范仅存在于语言的实现中。你可以考虑接受一项挑战：为它撰写规范。

如果语言的标准库是开源的，你可以采用“使用源码”模式中描述的技巧把它通读一遍。其中代码的质量不见得能给你留下较好的印象，但你要想到，代码的作者那时还没有一个社区提供给他向别人学习的机会，他们不得不摸着石头过河。你可以考虑给他们发送一个补丁来修正你所发现的bug，另外一种积累语言知识的方法是请教那些与你共事的人，问他们是如何选择自己所精通的首门语言的。然后把他们提供的准则添加到自己在选择入门语言时已经使用的准则当中。

最后，除了我们前面提及的“Muenchian方法”、“Schwartzian转换”和“Duff设备”，你还可以找到更多这样的惯用法（idiom）。它们都是以某个程序员的名字命名的，这些程序员都曾有一个具体的问题要解决，于是才发明了那种惯用法。试着去追查一下这些惯用法最开始试图解决的问题，然后考虑如何使用自己的入门语言来解决同样的问题。

参考模式

“质脆玩具”（第5章），“深入挖掘”（第6章），“找人指导”（第4章），“漫漫长路”（第3章），“使用源码”（第5章）。

白色腰带[译注3]

通常，每一步都该有进门的感觉。这是初学者的心——
一种正在“成为”的状态。
——铃木俊隆，《Zen Mind, Beginner's Mind》（禅者的初心）

情景分析

你已经深入理解了自己的第一门语言，舒适地享受着自信的状态。同事们发现了你的能力，请你帮忙解决属于你的专长领域内的问题。你为自己的技能感到自豪。

问题描述

你在努力学习新的东西，但新技能的学习看起来比以前更难了。尽管你已经尽了最大努力，但自学的速度似乎却在降低。你担心自己的个人发展就要停顿了。

解决方法

在对自己已有的技能保持自信的前提下，当接触到新情况时，你应该先放下已有的知识。正如Yoda在电影《The Empire Strikes Back》（星球大战2：帝国反击战）中所说的：“你必须忘记已经学到的东西。”（You must unlearn what you have learned）

系上白色腰带，是基于这样一种认识：系黑色腰带的选手知道该怎么做，而系白腰带的练习者别无选择只能去学习怎么做。

译注3：在许多武术门派中，束白色腰带的代表初级练习者。

作为一名家庭问题顾问，Dave采用的方法之一就是保持一种不知道的姿态。遭遇困境的家庭都是在经历一种特殊的情况，尽管接受过相关训练，但Dave知道自己不可能完全理解那种情况。虽然，在如何促进建设性的提问和交谈方面，他对自己的技能颇感自信，但他也曾学到：在那些家庭所经历的现实问题上，不要相信自己有任何专家知识。这听起来违反直觉，实际上却是在培养一种尊重和好奇的态度，这种态度能揭开意料之外的可能性和解决方案。Dave不会把解决方案给这个家庭，相反，他的不知道的姿态使他能跟这个家庭形成一个团队，大家一起合作找出解决办法。

采用这种方法来学习新技术会大大加速你的学习过程。训练自己暂停已经习惯的编程机制，这会使你发现新的可能性。然而，作为一名终将因自己获得高水平技能而感到自豪的程序员来说，向无知退一步并让自己看起来很傻会很痛苦。不过，你不妨考虑下George Leonard在《Mastery》（精通）的最后几页上说的话：

> 有多少次你担心被人觉得愚蠢而没有尝试新东西？有多少回你担心被人认为幼稚而压抑了主动性？……心理学家Abraham Maslow发现，那些能把潜力发挥到显著水平的人身上都有一种孩童般的品质。Ashleigh Montagu使用术语“婴儿化”（neotany）（来自词语“neonate”，新生儿的意思）来描述像莫扎特和爱因斯坦那样的天才。发生在朋友或自己身上的一些事，我们认为愚蠢，紧锁双眉；但同样的事如果发生在世界著名的天才身上，我们只会觉得古怪，一笑置之；永远不要忘记：可以随便犯蠢的自由很可能是打开天才成功之门的钥匙。

或者看看下面的例子，一位有10年行业经验的老手一直保持虚心并不断学习新东西：

> 我已经很专业很成功地编写软件长达十年之久，实践TDD也有好多年，直到有一天，我偶然翻到那本Michael Feathers编著、Prentice Hall出版、不幸被命名为《Working Effectively with Legacy Code》（修改代码的艺术）的书。这本书对我如何编写代码产生了直接而又深远的影响，我给我那个小公司的开发者每人买了一本，并要求他们阅读。从那

> 之后，我的代码库被逐渐改良成一个测试更良好、耦合更松散、适应性更好的系统，对它的开发和维护也带来了更多的乐趣。
>
> ——Steve Smith，电子邮件

正如Steve所学到的，我们必须能够放下过去的经验和先入为主，这样才能把新的知识放进来。在你尝试学习第二门编程语言时这尤其困难，因为它可能是你第一次为提高技能水平而牺牲生产率。先前你已经以“空杯心态”解决过一些问题，那时对于解决问题的“正确方法”你几乎没有任何先入之见。现在，在新知识有机会渗入到大脑之前，你必须设法避免将新旧知识混合在一起，并以初学者的心态着手解决新问题。这可能意味着短期内生产率会降低一些，以期在掌握新方法之后能获得技能水平的飞跃。

> 为了攀登高峰，你必须离开原来那个稳当的落脚处，放下已经取得的成果，甚至可能需要下滑到一个峡谷中。如果你永远不肯放下已经取得的成果，你仍可能不断取得平稳的进步，却永远不能抵达高峰。
>
> ——Jerry Weinberg，《Becoming a Technical Leader》（技术领导之路：全面解决问题的途径），第42页

在学习新的语言、工具或业务领域时，培养起这种心态有一种好处：你可以开放地学习如何用合乎惯例的方式表达自己，从而使自己和已有社区的沟通更加流畅。避免了“用各种语言写Fortran程序”的老问题，对新知识就会有更深刻的理解。这样，最终新旧知识融会贯通之时，你已经处在了更好的位置上，可同时从两个领域获得富有成效的见解。

以下面的Java代码为例。这段代码为英国的国家彩票产生随机投注数，方式是打印六个1~49之间的不同数字。

```
public class Lottery {
   private static final int NUMBER_OF_RANDOM_NUMBERS = 6;
   private static final int MAX_RANDOM_NUMBER = 49;

   public static void main(String[] args) {
       SortedSet randomNumbers = new TreeSet();
       Random random = new Random();
       while (randomNumbers.size() < NUMBER_OF_RANDOM_NUMBERS) {
```

```
            Integer randomNumber = new Integer(random.nextInt(MAX_RANDOM_
            NUMBER) + 1);
            randomNumbers.add(randomNumber);
        }
        System.out.println(randomNumbers);
    }
}
```

如果别人要求你用一种稍微不同的语言，比如Io（这种语言被设计成拥有尽量少的语法元素，同时又能被主流程序员接受）来重新实现这一功能，你可能复用很多Java知识并写下这样的代码：

```
list := List clone

while (list size < 6,
   n := Random value(1 50) floor
   list appendIfAbsent( n )
)

list sort print
```

但如果你被要求用一种完全不同的语言，比如J来重新实现它，你会发现这种方法不灵了。只有“系上白色腰带”，接受“在一种没有循环的语言中，肯定会有迥然不同但依然有效的方法来解决问题”这一事实，你才能取得进步。因此，按照J语言中的惯用法，答案是：

```
sort 1 + (6 ? 49)
```

后面我们将给出一些模式，来帮你练习，帮你设计玩具程序，并让你有意识地反思手头的工作。对于你的知识来，这些模式能使你更深入地理解不同部分之间的共性，并构造一些情景，让你能研磨自己的技能，而不必承受维持正常生产率水平的压力。

行动指南

寻找机会来忘掉一些东西。最好是迫使你放下以前经验的机会。

比如，找出一个采用某种编程范式（如命令式、面向对象、函数式、面向数组/向量等）编写的程序，然后用一种基于不同范式的语言来实现它。确定新实现遵循了新语言的惯用法。如果你所了解的所有语言都采用相同的范式（如面向对象），那这就是一次学习新范式的机会。

这种模式解决的不只是编程语言的问题，只不过在这方面编程语言是一个很容易产生误解的领域。你应该去请教一位使用你不所熟悉的编程语言或技术的人，让他帮你解释一下：有你这种特定知识背影的人，通常会对他们的社区产生怎样的误解。

参考模式

“质脆玩具”（第5章），“不断实践”（第5章），“且行且思”（第5章）。

释放激情

软件技师们只会雇佣对软件开发工艺
有强烈学习欲望的学徒。
……
学徒是软件工艺的重要组成部分，
因为它带来了能感染所有人的
学习激情和学习动力。
——Pete McBreen，
《Software Craftsmanship》（软件工艺）

情景分析

对于软件开发工艺，你有着无尽的兴奋和好奇心。

问题描述

你意识到自己对工作的热情远远高于你的同事，你发现自己正在抑制自己。

解决方法

尽管你缺乏经验（而且恰恰因为你缺乏经验），你会为团队带来一些独特的品质，包括富有感染力的激情。不要让任何人压抑了你对软件工艺的兴奋——它是一种宝贵的财富，它将加速你的学习。

作为软件开发者，你将不可避免地成为团队一部分，并在此基础上展开工作。在任何组织结构中，都会有一种遵循规范的趋向，对新人而言尤其如此。大多数团队都不会对技术有过热的激情。可以肯定，他们都专注于交付下一个项目，或者改善软件开发周期中的某些让他们头痛的方面。因此，充满激情的学徒常常会屈服于“做人要低调”的外界压力。他们要么完全压抑自己的热情，要么仅在日常工作之余才让它表现出来。

在相对完善的团队中释放激情当然是有风险的。如果团队士气较低或者团队不欢迎新人，你可能在背后遭遇白眼。对那些认为竞争力比学习能力更重要的人来说，毫无疑问，你会给他们留下不好的印象，特别当你暴露出自己无知的时候。跟任何模式一样，这一模式也不应被盲目运用。团队动态（team dynamics）永远都是需要考虑的因素。如果你发现自己处在一个无法接受你激情的团队中，那么你将需要想些办法来养护自己的激情。

然而，在一个对学徒的兴奋和贡献持开放态度的团队中，你将能为团队输入独特的品质，而且是有经验的开发者可以依赖的品质，比如自由的想象力和激情。在你的整个职业生涯中，只有这段时间冒冒风险，大声讲讲自己的想法是最靠谱的。你几乎不会因此失去什么。你的思想、热情将增加团队的智慧，并带来多样性。James Surowiecki在他的《The Wisdom of Crowds》（百万大决定：世界是如何运作的？）一书中多次指出：思想的多样性应看作集体智慧的关键因素。

对航空母舰舰队集体心理的一次有趣的研究显示：要安全地操纵一艘战斗机不断从它上面起降的巨船，需要复杂的、相互配合的行动，而新人在这类行动中扮演重要角色。研究者发现，一个由不同经验水准的人组成的团队更加健康。

> 如果把不同的经验水平关联到一起，比如当脑子里没有任何“想当然”的新手与认为自己已纵观全貌的老前辈们更加频繁地打交道时，大家对问题的理解都会加深。
>
> ——Karl Weick和Karlene Roberts，《Collective Mind in Organizations》（组织中的集体思想），第366页

最后，释放激情是学徒们相对较少的责任之一。你可能不会为团队带来深奥的知识和极高的生产率，但是，为团队注入一些兴奋因素，并对所有的事情表示疑问是你的责任。你（临时）处在一个独特的位置上，拥有新鲜的视角，这使你能提供一些有用的建议，从而改善整个团队。

> 学徒向技师学习，技师也向学徒们学习。激情洋溢的初学者不仅能恢复技师的活力，还能从外界带来新的思想来挑战更有经验的技师。精心挑选的学徒甚至能使师傅变得更有成效。
>
> ——Pete McBreen，《Software Craftsmanship》，第75页

行动指南

想想上一次你有一种想法却没有提出来是什么时候。找到那个你本想向她提出的人，向她描述一下自己的想法。如果她帮你指出一些缺点，试着说服她帮你改进。

参考模式

“暴露无知”（第2章），“培养激情”（第3章）。

具体技能

拥有知识，与拥有运用知识来创作软件应用的
技能和动手能力不是一回事。
这就是“工艺”发挥作用的地方。
——Pete McBreen，《Software Craftsmanship》

情景分析

你正打算到一个技艺更高超的技师团队中谋求一个职位，希望他们能提供比现在更好的学习机会。

问题描述

不幸的是，那个团队并不愿意冒险雇佣一名可能对团队没有直接贡献的人。而且还有另一种可能：你对团队甚至连间接的贡献都不会有。比如说，他们已经把简单的任务全部自动化了。

解决方法

你应该学会并持有一些具体技能。虽然学徒能把快速学习的能力带给团队，但在技能水平达到一定高度之前，在特定的工具和技术领域拥有具体的、可展示的能力还是会增大团队信任你，相信你能对他们有间接贡献的可能性。

在需要学会的具体技能中，有些无非就是一些能让你能通过HR过滤的办法，或者应付一些靠术语游戏来构建团队的经理们。另外一些技能则可以让未来的团队成员们对你放心，相信你能被很好地利用，而且不需要“日常照料”（《Organizational Patterns of Agile Software Development》（敏捷软件开发的组织模式），第88页）。具体技能的例子包括使用各种流行的语言编写构建脚本，了解像Hibernate和Struts这样的开源框架，基本的网页设计技能，JavaScript，以及你所用语言的标准库。

要点在于：你会经常需要“雇佣”一些经理，让他们对选择你有信心的飞跃。具体技能（最理想的情况是你可以带一个玩具应用的实现来面试）使你可以“迁就”他们。你的具体技能就是用来回答这种问题的：“如果我们今天聘用了你，周一上午你能帮我们做点什么呢？”对“入门语言”的深入理解能帮你建立信任度，对团队而言也是很有用的。

随着你慢慢过渡到熟练工的角色，你将变得越来越不依赖于这些技能，慢慢地别人开始基于你的名声、你以前参与过的项目，以及你能带给团

队的更深层品质来雇佣你。在这一天到来之前，你的优点必须更明显一点。

Dave填补空白

跟许多很晚才开始学习编程的人一样，我带着一大堆生活经验进入了程序设计领域，远远多于你们这些拥有6个月经验的一般程序员。从以前的职业中，我已经学会了各种人际交流技巧，也有了一些心理方面的感悟。在我作为一名程序员不断前进的过程中，我还是能遇到非常激动地谈起我过去的人，这使我偶尔高估了这些软技能，或者过于关注非技术主题了。的确，我身上的软技能曾经让我过得很好，并在很多情况下对我产生了极大的帮助；但是，为了让自己把大部分精力关注在技术技能方面——那显然是我最欠缺的领域，我那时不得不让这些技能稍微退缩一下。我改换职业并不是因为想成为程序员问题专家；而是因为我热爱制作软件的活动。

——Dave Hoover

行动指南

针对那些技能让你欣赏的人，收集一下他们的履历（Curriculum Vitae，CV）。你可以直接向他们要一份或者从他们的网站下载。对其中的每个人，找出履历上提到的五种具体技能，确定其中哪些是对你要加入的团队是直接有用的。总结出一份计划，设计一个可证明你掌握了这些技能的玩具项目。然后实施这份计划。

要养成定期把自己的履历审查一遍的习惯。一边看一边把具体技能提取到一份单独的列表中。你知道吗？许多雇人的经理只会看这份列表中的项目，而不愿看你的经验总结。

参考模式

“入门语言”（第2章）。

暴露无知

明天我要让自己看起来更傻一些，而对此的感觉要更好一些。
那种保持沉默并猜测到底发生了什么的作法是行不通的。
——Jake Scruggs在“My Apprenticeship at Object Mentor”（我在Object Mentor的学徒经历）中讲到的[注5]

情景分析

那些雇你做软件开发并为你支付薪酬的人，需要依靠你来了解你所做的事情。

问题描述

经理和队友需要对你有信心，相信你能够交付结果，但你对某些需要用到的技术并不熟悉。这样的事情不只会发生在咨询顾问身上。它会发生在所有人身上。你被带进这个团队，或许只是因为你对业务领域有深入了解，或者只是了解团队技术栈的某些其他方面。亦或者你是唯一可用于这项工作的人力了。

解决方法

向那些依靠你完成工作的人说明：学习过程是交付软件的一部分。让他们看到你在成长。

根据社会心理学家Carol Dweck的研究，在大多数工业化社会中，“让人觉得有能力”的思想深深地植入人们的意识中。随着软件日益渗透到日常生活的每个角落，作为开发者，社会越来越依赖于你具有这样的能力。然而，因为经验不足，你有许多无知的领域。你很盲目。你周围的人——你的经理、你的客户、你的同事，更不用说你自己了——都处在交付软件的巨大压力之下。当人们问你完成特性X需要花多长时间时，

注5：*http://www.jikity.com/Blah/apprentice.htm.*

你能从他们的眼中看到他们多么需要信心。你有很大的压力，想使他们安心，想打消他们的顾虑，让他们知道你很清楚他们想要什么，你将怎样、何时把结果交给他们。

软件技师通过与客户和同事的紧密联系来打造自己的名声，向难以描述的压力屈服，并对别人说他们想听的话并不是构建紧密关系的好方法。把真相告诉人们。让他们知道你已经开始理解他们想要的是什么，而且正在学习怎样把这样的结果交给他们。如果你想让他们安心，那也应该通过你的学习能力，而不是通过假装知道自己并不知道的东西。这样，你的名声将建立在你的学习能力上，而不是你已经知道的事情上。

暴露无知，最简单的方法就是问问题。但说起来容易做起来难，如果你要问的人以为你已经知道了答案，做起来就更难。别管那么多，坚持要问！当然，你可以保护自己的自尊，并通过不那么直接的途径来获得所需的知识，但不要忘了，如果采用最直接的可用途径，通往熟练工的道路就可以被缩短。经过时间和实践，直接去问团队里最明白的人会成为你的习性。在暴露无知的同时，你也向团队展现了自己的学习能力。而且，有时他们也能从回答问题的过程中对自己的知识获得新的认知。

一种“不知道”的姿态

> 作为家庭问题顾问，我曾了解到：要舍弃自己对别人的生活拥有专家知识的想法，采用一种“不知道”的姿态来接近人们。这是一粒必须吞下的苦药丸，不管你是一名新的家庭问题顾问还是一名新程序员。你的本能告诉你要掩饰自己的无知，装作通晓各种专家知识，但这只会阻碍你的成长，并阻止你完成正在尝试的工作。我把这种经验从一种职业带到另一种职业，这对我很有帮助。实际上，在日常工作中，我已经依赖于这种无知的感觉；它使我知道我正处在正确的位置。我正在成长。
>
> ——Dave Hoover

要习惯这种学习过程。这是一种技能。的确有一些对这种过程感到不舒服的人。这类人不会成为技师，而会变成专家（expert），即那种专长于某种平台或某一领域，并坚守这一平台或领域的人。由于他们的关注面更窄，这样的专家能在特定上下文中交付功能，而且会做得比其他人

都好。让我们这个行业拥有专家当然很重要，这也是必然的事情，但那不是学徒的目标。

专家技能是我们走的这条漫漫长路的副产品，但不是目的地。在旅途中，技师的工作会接触到数不尽的技术和领域。如果，因为必须或者因为兴趣，他们“深入挖掘”，并在一种或多种这样的技术方面掌握了专家技能，那就再好不过了。这是可以期待的，正如马拉松长跑训练会锻炼出更强壮的腿部肌肉。她不是在锻炼腿部肌肉；她是在锻炼跑步。就像一个行动积极的开发者，在一个Python项目上做了两年之后，会获得对Python知识的深入掌握，马拉松运动员健壮的腿部肌肉也只是一种手段，而非目标。

有些专家不遗余力地只与单一的上下文结合，收缩学习、实践和项目的范围。技师却恰恰相反，他们在拎起一门新技术或者学习一种新领域时，需要有勇气和谦逊来放下已有的专业技能并系上“白色腰带”。

技师所拥有的重要品质之一就是学习的能力，他们能找出无知的领域并通过努力工作来减少这样的领域。无知就像草地里的秃块，不断散播知识的种子，它就会减少。通过实验、实践和阅读来浇灌你的种子吧。你也可以将这些秃块隐藏起来，因为它们的面积让你感到窘迫，你想把它们遮盖起来以保持自尊不受影响。或者，你也可以选择暴露它们，对自己和依靠你的人保持诚实，并寻求帮助。

在学徒期结束时，你会深入掌握一些技术的丝线。凭借这些丝线，你可以在少数的平台和领域中编织出健壮的软件应用。师傅则可以用无数的丝线织出一幅锦绣。他无疑会拥有自己最喜欢的丝线和最喜欢的组合，但丝线的数目是巨大的，使得师傅能适应广阔的技术环境。这就是“漫漫长路”会把你带向的地方。相对于掩饰无知以显得自己有能力，暴露然后正视自己的无知能使你更快地纺起那根原本缺失的丝线。

行动指南

写下跟工作有关，而且自己不甚了然的五件事。将这份列表放到其他人

可以看到的地方。然后随工作内容的改变养成不断更新这一列表的习惯。

参考模式

“正视无知”（第2章），“深入挖掘”（第6章），“漫漫长路”（第3章）。

正视无知

如果我们看重独立性，如果在当前体系所包含的知识、
价值和态度方面，我们因为自己日益变得
从众而感到不安，那说明我们希望创造面向独特性、
面向自我导向（self-direction）、
面向自发精神的学习环境。

——Carl Rogers，《On Becoming a Person》
（个人形成论：我的心理治疗观）

情景分析

你发现了自己的技能空白，跟自己的日常工作有关系的空白。

问题描述

你需要掌握一些工具和技术，却不知从何开始。在这些工具和技术中，有一些是你周围的每个人看来都已经了解了的，而且别人认为你也了解。

解决方法

选出一种技能、工具或技术，积极地填补跟它有关的知识空白。

采用一种对你最有效的方法来做这件事。对于某些人，最好的方法可能是阅读能接触到的所有文献和FAQ，来获得知识概览。其他人则可能觉得直接动手构造一个“质脆玩具”才是理解一样东西的最有效途径。不

管哪种方法适合你，都不要忘了问问周围的“同道中人”和指导者，看是否有人已经掌握了这项技能并愿意分享所学。有时其他人也可能在学习这项技能，跟他们一起工作你会获得更快的进步。到某个时刻，你将在这个新领域达到令人满意的能力水准，这时你就可以做决定了，是继续深入挖掘下去更有成效呢，还是应将注意力转到其他的技能空白上。每天只有24小时，你无法将每项技能都研磨到很高水平，因此你必须学会在它们中间做必要的权衡。

这一模式与“暴露无知”紧密相关，但实施它的过程却不会对你的自尊形成那么多挑战，因为它可以私下完成，不会有人发现你不懂的东西。然而，作为一名有志于精通技能的学徒，你也要有“暴露无知”的意愿。单独运用这一模式（即正视自己的无知但不暴露它）是有风险的，那样会怂恿一种不接受失败和学习的文化，因为每个人都在秘密地学习。记住，公开地学习是学徒开始向熟练工跃进的途径之一。在人们能看到的地方学习，到可以向别人传授技能，其间只差一小步。

即使你成功运用了这一模式，也会有不好的副作用。如果因为想学习如何构造复杂的并行系统，你就用Scala语言[译注4]亲手写了一套消息系统，而没有使用现成的产品，维护你代码的程序员可能不会对此大加赞赏。如果他们无法向你提问，因为你正在参加一次学术会议，那他们就更不高兴了。最后，如果因为你的学习需要而妨碍了项目的成功发布，你的老板也不太可能谅解你。总之，你需要保持足够的警觉，不能让你的学徒期成为团队的问题。工艺方法的一个重要方面就是要有一种意愿：把更大的团队利益置于个体利益之上，而不是利用团队和客户来促进个人成长。

另一方面，也有可能你正视了自己的无知却没有“暴露无知”。如此行事的人在面对自己的无知时只会无辜地耸耸肩，像是在说“就是这样。”这会导致一辈子的谦卑、无知，并过度依赖团队中的其他成员。最后，它会导致这样的团队：每个成员都维护着自己的一丁点知识储备，当问题延伸到另一个人的知识领域时就耸耸肩。

译注4：一种多机制的编程语言，支持函数式和面向对象编程。

因此，在该模式和“暴露无知”之间精心地维持平衡非常重要。仅仅正视无知会导致自负的食知动物（infovores），最终一事无成；而仅仅暴露无知却不视之为需要解决的问题则会导致过度的谦卑和无能。

行动指南

针对“暴露无知”模式中列出的项目，努力学习其中的每一项，每学会一种就把它从列表中划掉。这些新的知识又会揭示你以前没有注意到的新空白；别忘了把这些新发现的空白也加入列表中。

参考模式

“质脆玩具”（第5章），“暴露无知”（第2章），“同道中人”（第4章）。

深水区域

如果你从来没有一败涂地，
那很可能你也没尝试过有价值的东西。
——Christopher Hawkins，“So You Want To Be a Software Consultant?”
（你想成为一名咨询顾问吗？）[注6]

情景分析

迈着安全的小步子前进已经无法让你满足。你开始担心这不是一片高原，只是一段泥沟（rut）。在高原上，你可以通过勤奋的实践来巩固自己的技能，从而达到更高的水准；在泥沟中，平淡无奇的能力终将衰落成碌碌无为。

问题描述

你需要提高技能和自信，为自己增加成功项目的数量和多样性。你觉得

注6：*http://www.christopherhawkins.com/08-30-2006.htm.*

有必要用更大的事情来挑战自己。包括更大的项目、更大的团队、更复杂的任务、新的业务领域，或者去新的地方。

解决方法

跳进深水区域吧。“一直等待直到自己准备好”会变成一张啥也不做的处方。因此，当你被安排扮演更高姿态的角色或者解决更加困难的任务时，要用双手抓住机会。只有承担艰巨的任务并做一些让人紧张的事情，你才会成长。

这样做也有风险。如果估算失误，水位高过了你的头顶，那你可能溺水。谢天谢地，在IT行业，有许多领域你可以冒冒风险，即使失败了也不会损害你的职业生涯。本来是机会，从半闭着的畏惧的眼睛里望去就成了风险。但这并不是教你在简历中撒谎来骗取一份本来无法胜任的工作，也不是让你在没有充分准备的情况下迎接挑战。它意味着当提升或者外部的任务被安排到你头上时，即使面临着极高的失败率，也要勇敢地接受它。为失败做好准备并从失败中振作将为你打开怯懦者永远看不到的大门。

双脚跳入

我来到这家做这类电信服务交付平台的西班牙公司。我加入的是核心团队，做一些常规工作，没什么挑战性。公司曾经换过一任CTO，而我对整体的方向就根本不满意。

我厌倦了，而且下定决心离开这家公司；正在这时，我听说公司要在尼日利亚安排一名咨询顾问。在一次厨房闲聊中，我向他们提及了自己愿意去的想法。

CTO和CEO找我谈话，问我是否愿意接受这种改变。我的合同将会调整，我将不再是一名正式员工。

他们想让我销售公司的产品，但我并不是一名销售员。然后我读到了一些吓人的报道，说拉各斯（Lagos：尼日利亚首都和最大城市，位于尼日利亚西南部的几内亚湾畔。）是世界上第三大危险城市。这真的让我感到恐惧。但我对自己说，如果真这么恐怖，我大不了当天再飞回来。

两周后我飞到那里。离开之前，我跟一个在那里生活过的同事聊了聊，这对我很有帮助。或许是我比较冒失或者愚蠢吧，但那些担心确实消失了。或许不只是前面几天，而是连续好多周，我都是如鱼得水的感觉。

本来计划的是三个月的短期合同，可我在那里呆了整整两年，一直帮助我们的客户。我意识到自己根本不可能在那里销售我们的平台。他们需要的是一些不同的东西。我们的平台对他们不会有帮助，于是，我投入精力来做这件事情，修正了他们的服务，并为他们构建了一套真正适合其需要的平台。

自从去了尼日利亚，我后来几乎跑遍了西非的所有国家。现在我在伦敦工作。

——Enrique Comba Riepenhausen，电子邮件

虽然我们提倡你去尝试自己所能承担的最具挑战性的任务，但你仍要记住：如果水位高过了头顶，你将会溺水。即使在Enrique的例子中，他让生活做如此重大的改变，但去的地方至少也有个熟人，而且那里还能说母语。你有责任对这种方法的风险做些调整，方法是参考“找人指导”和“同道中人”模式，找到在你需要时能提供帮助的人。

你也有责任“建立馈路”，从而当挑战性的项目失去控制时，你还可以抓住它并马上获得帮助。运用这一模式的时候，应该让人觉得你很勇敢，而不是很冒失。

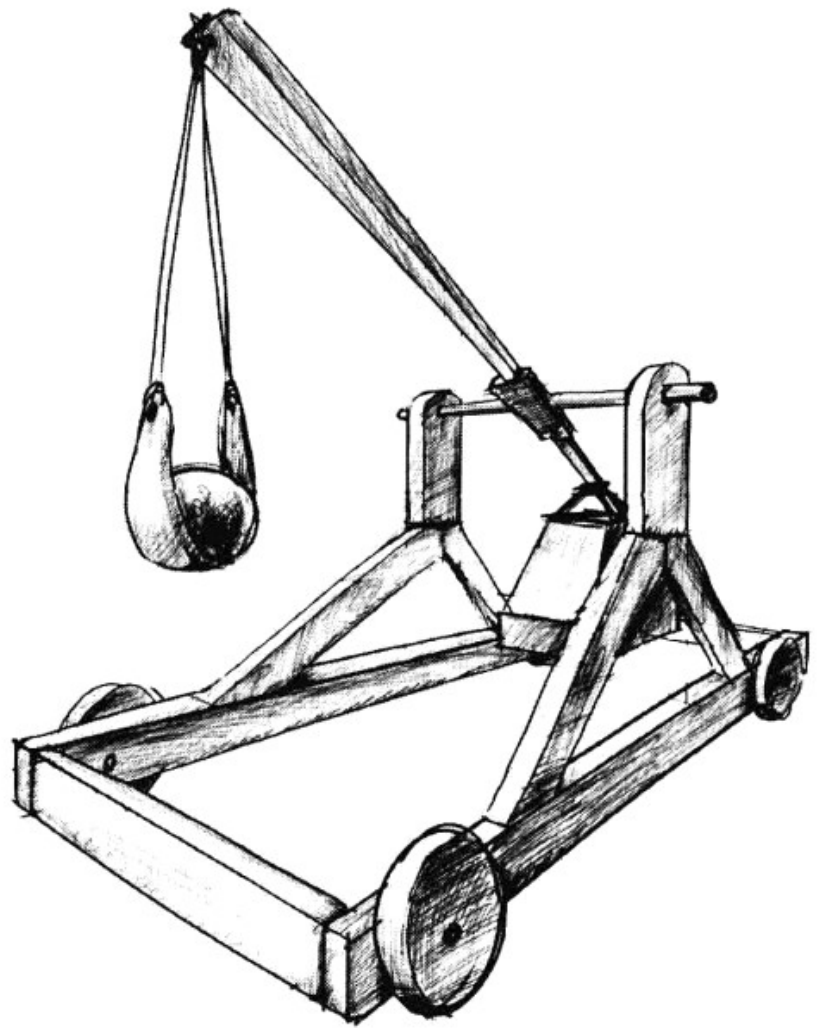
行动指南

按代码行数或开发者的数量来算，你做成功过的最大项目是什么？你独立构建过的最大代码库又是什么？写下这些问题的答案，然后看看能否找出项目复杂性的其他尺度，以及度量项目的其他方法。用这些尺度度量一下自己参与过的每个项目。现在，当下一个项目到来时，你可以把所有的项目画一张图，并找出新项目在其中所处的位置。一段时间之后，你将通过这张图看到自己职业前进的方向，甚至基于这张图来做决策。

参考模式

“建立馈路”（第5章），“找人指导”（第4章），“同道中人”（第4章）。

以退为进



“众里寻她千百度，蓦然回首，那人却在灯火阑珊处。”

沿着你蓦然回首时发现的模式向前推进，

有时你就能想出新的东西了。

——Robert Pirsig，《Zen and the Art of Motorcycle Maintenance》

（万里任禅游）

情景分析

你开始意识到自己的知识是多么匮乏，或者你已接受了新的挑战，但事情并不如意。也可能两者都有。

问题描述

当你发现了自己的大片无知领域时，感觉自己要倒下了。

解决方法

退一步，然后再向前冲，就像弹弓打出的石头那样。暂时退回自己的能力范围之内来获得一点镇定。花点时间构建一样你知道如何构建的东西。然后基于这种经验来认清自己已经走了多远，当前的能力有多高。

学徒期的体验就像坐过山车。你会体验到学习新技术的刺激，还有借助自己的知识和创造力向客户交付价值的刺激。然而，比起一路上遇到的成熟技师和专家，你开始觉得自己的知识是多么匮乏，因此你也会体验到那种心脏跳到喉咙的紧张。这会把人压垮，特别当最终期限已经临近，或者当你在处理产品问题的时候。振作一点。这是“漫漫长路”上正常的、不可避免的现象。战胜能力不足的恐惧正是“暴露无知”和“正视无知”之间的桥梁。

这一模式跟那些将自己过度拉伸到能力之外的人最有关系。如果你在学徒期基于合理的步伐、逐步增加的责任和技术复杂度前进，那么你不需要接受这一模式的庇护。但如果你真的在挣扎，或者说在深水区域你难以让自己的脑袋浮出水面，那你就应该寻找暂时回退的机会。有时你需要后退一步才能前进两步。这时，要尽可能快地将这种后退的动作转变成前进的冲力，这一点极其重要。这种前进的冲力将表现为你拥有了比昨天更丰富的知识和更高超的技能。

然而，向后退步也使这一模式的使用带上风险。如果不能清醒地选择一个后退的距离，你会发现，自己不过是向失败的恐惧投降了。更深地投入到已经有能力做好的事情中，会让人非常宽慰。专长带来的回馈是直接而切实的，但这样做的风险不会立即浮现出来，直到有一天你采取任

何措施都为时已晚。当你的专长最终变得过时，你将被迫再次面临一大片无知领域，而此时你可能已经失去学习新东西的习惯，重新开始会比原来痛苦得多。在这种情况下，克服那种被压垮的感觉比解决问题本身更困难。

要避免这种情况，你必须接受这一事实：该模式只是你聚集力量东山再起过程中的一个短期修正。为自己设定一个时间限制（或者说一个“时间盒”[timebox]），比如“在优化提供数据的SQL查询之前，我将把接下来的10分钟用于重构这个页面中的JavaScript验证。”或者“在学习如何调用这个第三方的SOAP API之前，我将把接下来的4小时用于实现这个工具的命令行界面。”或者“在开始优化那些Python‘全局解释器锁’（Global Interpreter Lock）影响到的代码之前，我将把今天剩下的时间用于提高测试的覆盖率。”

该方案的另一个重要方面是：利用短暂的休息向聚集在你周围的指导者和“同道中人”寻求支持。有了他们的支持，再加上最近一次展现能力的推动，你可以准备得更充分，从而能够应付再次尝试时前进道路上的碰撞。

行动指南

选择一项你已经非常了解的自包含任务，重新实施它。比如，Ade喜欢实现缓存算法，因为它可以非常简单，也可以超级复杂。这给他提供了机会，使他能强化自己在设计和算法复杂度方面的直觉。

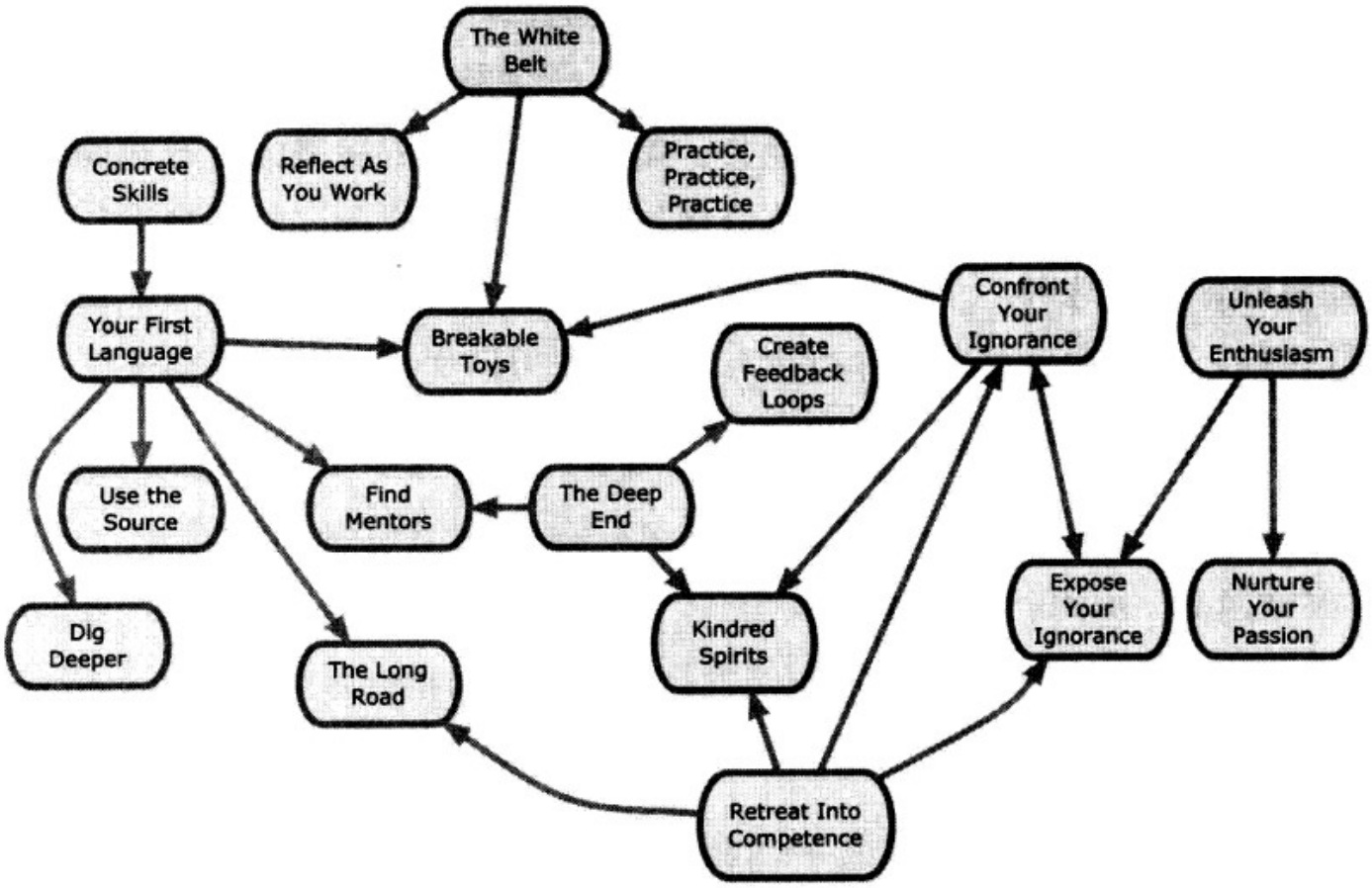
参考模式

“正视无知”（第2章），“暴露无知”（第2章），“同道中人”（第4章），“漫漫长路”（第3章）。

总结

本章讨论的“无知”、“深水”、“暴露”、“回退”，听起来都比较消极。但无知并不是一件坏事，只要你能意识到并正视它。最坏的情况

是你根本不知道自己无知，如果你能意识到自己缺少的东西并补充它，你就向前迈进了一步。要获得扎实的学徒期锻炼，基础之一就是“准确的自我评估”，在评估中你试着确定自己已经走了多远，并记下知识中的空白。你需要对自己已有的能力、需要赶快胜任的技能以及有长期兴趣的知识都了然于胸。后续章节谈到的一些模式，比如“且行且思”，能够帮你维持这种“了然”。最重要的是，要拥抱“漫漫长路”中的这段时间。在职业生涯中，你很少有其他时间可以保持如此的内向关注，并致力于个人的成长。


The White Belt
Concrete Skills
Reflect As You Work
Practice, Practice, Practice
Your First Language
Breakable Toys
Confront Your Ignorance
Unleash Your Enthusiasm
Create Feedback Loops
Use the Source
Find Mentors
The Deep End
Dig Deeper
The Long Road
Kindred Spirits
Expose Your Ignorance
Nurture Your Passion
Retreat Into Competence

第3章

走过漫漫长路

它不只是征服先前未知的高峰，
还要一步一步地走出一条新的道路。
——音乐家、作曲家Gustav Mahler

你有没有把培训或成绩的证书挂在格子间的四周？当Dave刚拥有自己的格子间时（那时他的经验比现在还要少），他在自己桌子的旁边刻意摆放了一大堆证书。这堆证书中比较有特色的是Brainbench公司的Perl“大师”证书，此外还有其他一些证明他完成了各种多日培训的证书，诸如C、J2EE、Vignette和ATG Dynamo方面的。这一堆伪文凭使他（以及他的公司）可以相信他知道自己在做什么。毕竟他被“培训过”了。

同时，Dave开始学习其他东西，并通过*http://perlmonks.org*和*comp.lang.perl.**新闻组与更广泛的开发者社区联络。正是在这些组织中，他发现了一些了不起的Perl黑客（hacker）[译注1]。这些黑客们的专业水平让人敬畏，尤其当Dave看到他们还在学习，而且快速地学习时，感觉尤其如此。这让他渐渐明白：真正意义上的“杰出软件开发者”，他连边儿还没摸到呢。在接下来的几个月里，它的那一堆证书就慢慢消失在更大的一堆便笺和打印出的书稿、教程下面了。

通过观察，并且与几个这样的杰出黑客交流，Dave被这一学习过程迷住了。每隔一段时间，他就会去看一看这些黑客们所学知识的深度和广度，然后垂头丧气地或者欢欣鼓舞地离开——垂头丧气是因为自己知道的是多么少，欢欣鼓舞是受这些黑客能力的影响。他开始投入到一些非正式项目中，并开始阅读自己能搞到的任何书籍。

随着Dave学到的更多，他也更认清了自己要走的路还有多远。在后来的几年里，他有幸同一些杰出的软件开发者面对面地合作。Dave看到，虽然这些了不起的人远远走在了他的前面，但他和他们却是走在同一条路上。

译注1：在计算机技术发展的早期，“hack”、“hacker”等词汇并没有任何贬义，它们仅指能熟练使用电脑或者用电脑编程的人。后来才慢慢被用来指代那些采用非法手段进入或接近别人的电子系统并窃取信息或搞破坏的人。

漫漫长路

一位很有潜质的学生问道："掌握合气道[译注2]
需要多长时间呢？"最为巧妙的回答是：
"你期待自己能活多长时间呢？"
——George Leonard，《Mastery》（精通）

要在编程方面真正擅长需要一生的努力，
还有不断学习实践的进取心。
——Ron Jeffries等，《Extreme Programming Installed》（极限编程实施）

每当你向着技艺精通的目标迈出一步，
你的目标就会向着更远的方向移动两步。
要把臻于精通的目标作为一生的努力来拥抱。
学着爱上这个旅程。
——George Leonard，《Mastery》（精通）

情景分析

我们生活在一个看重一夜成名、后起新秀、物质财富和快速回报的文化中。周围很少有程序员能告诉你在更早的年代里软件开发是什么样子的。如果你有机会跟那些老人们聊一聊，他们只会对最新的行业时尚摇头，因为那些东西只是在重复他们年轻时见过的错误。这些经验看起来都被遗忘了，因为能在一代代软件开发者之间留传下来的知识太少。

问题描述

你渴望成为一位软件师傅，而你的抱负同人们的期待不一致。传统的智慧告诉你：要取得薪酬最高的职位，并抓住你能获得的第一次提升机会停止编程工作，转到更重要的工作上，而不是慢慢培养自己的技能。

解决方法

首先，你要接受这样的事实：因为你的理想，别人会感觉你有点奇怪。

译注2：日本的一种徒手自卫术，利用对方的力气取胜。

然后，让自己关注更长远的目标。在学徒期，相比薪水和传统意义上的领导能力，你要更看重学习和长期成长的机会。

渴望成为软件工艺师傅的人们需要做长期的规划。这一长远（但光明）的旅程将带给你丰富的技能。你将精熟于学习、解决问题，以及同客户建立牢固关系的技巧。你将能娴熟地运用自己的知识和技术，就像一名侠客使用它的短刀和长刀。你将理解和领悟软件开发的深层真理。但所有这些都需要时间。

你应该对这一旅程的漫长有所准备。当你“自定路线”时，应该牢记自己期望成为一名开发者，即使到中年仍然一直工作的软件开发者。让这一期望来影响你的工作选择，以及抱有雄心的领域。如果你还想继续工作20年，那你就可以做任何事。考虑到磨炼技艺所花的时间，没有哪个人的技能水平是你跟不上的。也没有哪种业务领域或技术领域是对你关上大门的。考虑把整个职业生涯贡献给自己的技艺，那么超越像Donald Knuth或Linus Torvalds那样的人是现实的而不是虚幻的。旅程的漫长恰恰增加了向你敞开的可能性。（当然，像Knuth和Torvalds这样的人是不会原地不动等着你去追赶的！）

这一模式不是为那些渴望成为CIO或者项目经理，或渴望非常富有的人准备的。一路上，你并非没有可能去扮演权力和责任角色，或发现自己已非常富有。然而，这些角色和利益不是成功学徒的主要目标——他们只是终生旅程的副产品。技师们不会数日子等退休，他们会开心地工作，直到最后几年。

我们不想让你产生“所有人必须走同一条路”（参见“自定路线”）或者“这是适合所有软件开发者的正确路线”（参见“另辟蹊径”）这种印象。有些人永远地离开了开发工作，做起了主管、测试、销售或项目经理。有些人永远地离开了技术，进入完全不同的领域。这都没有问题，也都是可以选择的有益路线，但这本书和这一模式却不是针对这些人的。

如果说“准确的自我评估”是学徒期成功的角石，那“漫漫长路”就是

地基。如果说通往技艺精通的道路需要走很多步，那么从学徒到熟练工的转变只是其中的第一步。正像功夫高手争取黑色腰带级别的过程，新的熟练工也清楚自己还要走多远。

软件开发者是幸运的。属于我们的是一条博大精深的路，一条天生不断变化的路。摩尔定律继续无情地向前推进，不断地为技师们打开新的机会，可以探索新的平台，或者为已有的程序重新排列各项特性的优先级。而其他的变化常常是表面的。新技术替代了旧技术，解决的基本问题却都一样。虽然总会有新的软件需要学习，也总会有更好的硬件出现，“漫漫长路”教给技师们软件工艺的深层原理，使师傅们可以超越特定的技术，直击问题的核心。

行动指南

闭上眼睛，想象一下在10年内你可能扮演的最奇怪的角色是什么。开心地为自己设想一个最古怪的未来。然后想象一下从现在开始20、30或40年以后的情况。到那时你希望自己都尝试过哪些经历？设想40年后，有人让你对自己的职业历史和一路上对你影响最大的事情做个简单描述。利用这一思考练习的结果来帮自己规划未来的职业选择。

参考模式

“另辟蹊径”（第3章），“自定路线”（第3章）。

技重于艺

我愿意将编程说成一种技艺，技艺本身也是一种艺术，
但它不是美术。技艺的意思是：使用可能带有装饰性的手法来制作有用的对象。
美术的意思则是制作东西纯粹为了使之美丽。

——Richard Stallman在“Art and Programming”（艺术与编程）中[注1]

注1：John Littler，“Art and Computer Programming”（艺术与计算机程序设计），参见：*http://www.onlamp.com/pub/a/onlamp/2005/06/30/artofprog.html*。

情景分析

老板付你工钱是让你构建能为客户解决问题的东西。

问题描述

虽然存在已证明有效的方案，但要考虑到：客户的问题可能代表一次机会，你可以利用它来做一些真正奇妙的事情，即为自己提供机会创造一些漂亮的、会给同事们留下深刻印象的东西。

解决方法

技能建立在牢固的关系之上。相对于增加个人利益，要更重视向客户交付价值。

作为技师，你的首要工作是构建能满足他人需要的东西，而不是沉迷于艺术展现。毕竟，世上不该有挨饿的技师。正如我们的朋友Laurent Bossavit所说：“作为技师，挨饿是一种失败；技师应该有能力用自己的技艺谋生。”[注2]工作中你需要全力以赴，将客户的利益置于展现自身技能或充实履历的愿望之上，同时维持软件社区中形成的最低能力标准。“行走漫漫长路”意味着你必须平衡这些相互矛盾的需要。如果你挨饿了，因为你太像一个艺术家，你创造的东西太美以致在现实中无法交付，那你就是离开了技艺。如果你完成漂亮工作的愿望迫使你离开专业的软件开发，离开为真实的人们制造有用工具的活动，那你就是离开了技艺。

我们为客户构建的东西“可以”是美的，但“必须”是有用的。由这一模式所构筑的成熟过程，其中的一部分就是培养在必要时牺牲美丽来换取效用的能力。

沉溺于制造漂亮但无用的物件不是技术。相比一个基于100万行代码、推动了计算机科学前沿，但却不能玩的游戏程序，技师们更看重只有50行代码，但能博人一笑的游戏。

注2：　Laurent Bossavit，私人交流。

技重于艺的另一方面是：客户需要你产出令人满意的质量，即使你不喜欢这点。技师不会等到灵感扣动心弦时才交付让客户满意的产品。这既有积极的一面，也有消极的一面。一方面，软件技师被隔离在田园般的艺术领地之外；在领地里面，会有人付她工钱让她制造自己喜欢的东西。另一方面，因为制造并使用直接提供价值的软件，她和客户都感到满足。

Ken谈工艺技术

工艺的磨炼在于为真人解决实际问题，而不是只为自己踌躇满志。

——Ken Auer，电子邮件

这一模式并非告诉大家只做一时便利的事情。它也包含了这样的思想：一件有用的工艺品至少能表现出最低程度的质量。使用这一模式时，你必须在两件事上取得平衡，一是客户期望提供针对问题的直接解决方案，二是你成为技师总要有个内在标准。要做到即便处于压力之下也能坚持这些标准，你需要培养一种观念：功用和美好并不对立，而是相辅相成。一款软件越有用，软件拥有高质量就越重要。但质量需要时间来保证。你必须不断地在美好和功用之间折中，从而向着适当的质量级别前进。有时你会做出错误的折中，而将系统推翻重写来修正问题的方式不见得符合客户的最大利益。在这种情况下，你必须培养重构和修补的能力。正如Sennet所写：“正是在修正问题的过程中，我们得以理解事物的内部机制。”[注3]在这里，当你的航程太偏，离艺术和便利都已经太远的时候，花点时间修正航向能让你学会一些其他方式都无法教你的软件开发经验。

行动指南

在接下来的24小时中，找机会做一点有用而不是漂亮的事情。这可以是一个简单直接的选择，也可能需要更加微妙的权衡。重要的是：在你选择做什么的时候，确保自己已经意识到这里讨论的问题。

注3：《The Craftsman》（工匠），第199页。

要让自己更了解这些问题，另外一种方法是：回想一次去年发生的场景——那一次你选择了艺术而不是功用。当时的结果是怎样的？如果你当时做了不同的选择，结果又会怎样？把这些想法写下来。

参考模式

“漫漫长路”（第3章）。

持续动力

任何人，只要他见过工作中的程序员……都明白：
如果程序员有机会按照自己喜欢的方式来编程，
那编程本身就是编程的最大动力。
——Jerry Weinberg，《The Psychology of Computer Programming》
（程序开发心理学）

情景分析

作为学徒，你必须发展自己的专业技能。因为这个，你常常发现自己正埋于含糊不清的项目，处理着其中凌乱不堪的现实，面对着客户不断摇摆而且相互冲突的需求。

问题描述

在软件开发的阵地上致力于真实项目开发不是那么容易的，有时感觉沉闷乏味，有时让人疲惫不堪，常常让人灰心丧气，并频繁出现过度的混乱和拘束。

解决方法

要确保你对软件工艺的动力能够适应环境，并度过“漫漫长路”的考验和折磨。

在几天、几周或几个月中，你会热爱自己的工作。你会笑出声，崇拜自己真正通过开发软件而赚到了钱。你所编写的软件会毫不费力地从你的

大脑中流淌到你的指尖上，功能和设计看起来都很美。这些都是好日子，非凡的日子。换言之，它们不是你的平常日。

> ……有一类软件可以赚钱，有一类软件写起来有趣，这两类之间却没有太多的重叠……如果想挣钱，往往就被迫去解决那些非常难处理、没有人能轻松解决的问题。
>
> ——Paul Graham，《Hackers & Painters》（黑客和画家）

Paul Graham说得恰如其分，一般的编程工作会使你直面那些沉闷乏味、定义模糊、而且复杂得让人觉得多余的问题。这些都是乱七八糟难以处理的问题。此外，你还会面对官僚作风和难对付的人，并接受不规范的领导。你会在几天、几周或几个月的时间里怀疑自己对开发的投入。当遇到这样的问题，让自己编程的动力向着行走“漫漫长路”的目标看齐就显得至关重要。下面的几个例子能说明这点：

- 你厌恶自己的编程工作，你的主要动力来源于金钱。你发现自己更专注于攀爬企业的阶梯，而不是磨炼自己的技艺。但是你也有一部分动力来自于技术专家的名声，这使得你能够忍受足够长的时间来等待工作局面的改善。
- 你的主要动力来自于享受编程的乐趣，但已经有好几个月你感受不到对编程的钟爱了。你在严肃地考虑放弃这一职业。幸运的是，你也有一部分动力来自于金钱，你觉得从收入上讲，编程工作是你目前最好的选择。你为了金钱一直坚持，最终你对编程的钟爱又回来了。
- 你致力于开源项目的开发，主要动力来自于建立名声的愿望。当你的项目为遍及全世界的用户提供了有用价值，你的黑客地位却不再前进了，你开始考虑放弃自己的工作。然而，对免费软件重要性的坚定信念使你继续前行。最终你的项目遍地开花，自己的名声也上去了。

有些程序员一不小心就掉进了自身动机的陷阱里。在《More Secrets of Consulting》（咨询的奥秘——咨询师的百宝箱）一书中，Dorset

House和Jerry Weinberg将这种现象描述为“金锁”：“我想学一些新东西，但已知的东西能让我获得很好的回报。”“金锁”风险显示了让自己的动力向“漫漫长路”目标看齐的重要性，你需要有雄心有抱负才能达到对技艺的精通。金锁的出现是不可避免的，对技艺精通的向往会促使你提防它们。作为一名技师，随着不断取得进步，你将面对艰难的抉择：决定你是否能继续自由地行走在“漫漫长路”上，或是否会发现自己已掉进金锁的陷阱里。有两个例子：

Obie Fernandez是一名杰出的Java程序员，他的名声就建立在他的Java技能上。Obie在2005年面临一个决定：继续提升自己Java专家的身份，还是基于一种不熟悉的语言（Ruby）来学习一种有希望的新Web框架（Rails）。Obie选择了学习新语言，以此来扩充自己的工具集。这就是技师的标志。他把自己Java名气的安全性放在一边，成为了一名Ruby新手，避开了金锁。有意思的是，相比之前的Java专家地位，这一决定使Obie登上了更高的高度，最终使它创建了Web应用开发公司Hashrocket。

有好几次，Marten Gustafson发现自己正走在项目死亡行军（project death march）的半路上，因为他对技艺的热情使他把自己的全部时间和精力都投入到项目中了。出于个人英雄式的扭转乾坤的良好愿望而将自己扔进这种无底洞的年轻人，Marten不是第一个，也不会是最后一个。当你行走在通往技艺精通的“漫漫长路”上，在软件技艺上“培养激情”，并使之与生活的其他方面保持平衡是非常必要的。有时天平会倾向一边或另一边，这是很自然的。但是，在整条“漫漫长路”上，你都应该对这种平衡动作保持清醒的意识。

Dave的低痛阈（low pain threshold）

在这一模式的演化过程中，David Wood（从我在ThoughtWorks的日子开始一直到现在的“同道中人”之一）向我分享了一些传统智慧：“做你喜爱的事，钱自然会来。”这一建议甚合我意，因为当我不能做自己喜爱的事情时，我就完全是个懦夫。另一方面，当我做自己喜爱的事情时，我就会发现自己有不可阻挡的创造性和活力投入到工作中，最终为

我带来更多的收入。虽然许多程序员可能在短期内找到收入更高的工作，但做自己喜爱的事所带来的收入从长期来看也将是非常丰厚的回报。要阅读关于这一传统智慧的更多内容，可以去翻翻Steve Jobs（不是别人，就是那个退学生和Apple的共同创办人之一）在Stanford大学2005级学生开学典礼上的演讲。[注4]

——Dave Hoover

行动指南

写下至少15项能为你提供动力的事情。稍等一会之后，再写下另外的5项。你有多少动力来自其他人的想法而非自己的感觉？它在前面15项和后面5项中分别占据的比例一样吗？有多少动力因素是可以不要的？现在列出5项最重要的为你提供动力的事情。把这份列表保存到某个地方，遇到困难时可以看看。

参考模式

“培养激情”（第3章），“漫漫长路”（第3章）。

培养激情

上帝只让一小部分人开开心心地通过自己喜爱的工作谋生。

感谢上帝，让我成为其中之一。

——Frederick Brooks，《The Mythical Man-Month》（人月神话）

情景分析

老板雇你来，“只”当你是一名软件开发者。

问题描述

你工作在一个不好的环境中，它扼杀你对软件工艺的激情。

注4：*http://news-service.stanford.edu/news/2005/june15/jobs-061505.html.*

解决方法

采取措施来保护并培养自己对软件技艺的激情。

要成为熟练工，你需要拥有对软件技艺的激情。不幸的是，你每天的活动常常使这种激情减退。你可能面对着混乱的企业等级制度，向着死亡前进的项目，满嘴脏话的经理，或者玩世不恭的同事。处在这种恶劣的环境中，你很难培养自己的激情，但你可以采取一些基本的行动来维持它。

做点自己喜欢的事情。从工作中找一些让你感兴趣的东西，确定它确实是你喜爱的，然后让自己投入其中。如果工作中你不能腾出足够的时间来做这事，考虑安排一点额外的时间。如果这不可行，抽点工作外的时间来构造一些“质脆玩具”吧。

在命名为“伟大黑客”（Great Hackers）的2004年O'Reilly开源大会（O'Reilly's Open Source Convention，OSCON）上，Paul Graham做了一次演讲，他说：“成为伟大黑客的关键在于做自己喜爱的事……要把一件事情做好，你必须热爱它。所以，只要你能坚持对编码的热爱，到了这种程度，你就会做得很好了。”

找一些“同道中人”。加入一个关注你要深入学习的某样东西的本地用户组。开一个博客，阅读一些你感兴趣的博客。参加在线论坛和邮件列表来“分享所学”。启动一个使用“知识消防栓”（Knowledge Hydrant）模式语言的学习小组，这一模式来自Joshua Kerievsky的论文“A Pattern Language for Study Groups”（针对学习小组的模式语言）。[注5]

“钻研名著”。当你的激情陷入危险处境时，可以让自己沉浸到软件开发领域的杰出作品中，这可以帮你越过难关。这些永恒的书籍能打开你的视野，让你看到不同的世界，一个一切可以变得更好的世界。

注5：*http://www.industriallogic.com/papers/khdraft.pdf.*

“自定路线”。有时你的需要、目标和抱负会跟老板提供给你的职业道路相抵触。可以考虑换到一个新的、能提供适合自己的职业道路的组织，这样可以保护你的激情。

项目死亡行军（project death march）是最具破坏性的恶劣条件。当面对向着死亡的行军时，很难想象如何保护自己的激情，更不要说培养它。它会耗尽你的时间和精力，使你不能采取任何有效行动来保护自己的激情，因为有更重要的问题——比如个人的健康状况和紧张的家庭关系——需要你去关注。死亡行军非常迎合许多软件开发组织中流行的个人英雄主义思想。行走在“漫漫长路”上的人们并不是短短几年冲刺然后燃尽自己的英雄——他们是在几十年的时间里保持可持续的节奏，并不断前进的人们。

要培养自己的激情，需要设置一个清晰的界限，基于这个界限来定义你愿意身处其中的工作环境。这或许意味着你早早下班而团队的其他人加班到很晚，你退出一次恶言相向的会议，你将一次玩世不恭的谈话导向充满建设性的议题，或者你拒绝分发没有满足自己最低要求的代码。结果可能是你在加薪升职名誉声望方面被忽略了。但是，如果你希望打破恶劣的条件并保持自己的激情，这些界限就是必需的。

Paul Graham在他的OSCON演说中继续说道：“要保持你在14岁时对编程所抱有的新奇感。如果你担心现在的工作正在让你的大脑腐烂，那它很可能确实是。”

> 有些旅行者的职业、生活与其通往技艺精通的道路是吻合的，他们是幸运的；其他人必须在常规的工作时间之外找到时间和空间做点自己喜欢的练习，这会带来精湛的技艺却不会提供生活的薪水。
>
> ——George Leonard，《Mastery》（精通），第133页

行动指南

在上班路上准备三个可用于讨论的积极想法。在一天当中，如果某一次交谈开始侵蚀你的精力，就把交谈的内容导向你备好的议题之一上去。

目标是接过控制权，避免被周围的负面交谈拖垮。在回家的路上，回顾一下自己成功的程度，并考虑一些改善环境的其他办法。

参考模式

“自定路线”（第3章），“同道中人”（第4章），“钻研名著”（第6章），“漫漫长路”（第3章）。

自定路线

要小心那些不靠谱的批评家。在社会的某些场合，
我们可能遇到一些同事或者一些人，他们试图证明：
随着时间过去，编程……会变成一种不可持续的活动。
他们认为软件开发只适合年轻的毕业生……而当我们结婚
并有了小孩之后，便不能再做软件开发了。
——Mohan Radhakrishnan，闲谈[注6]

情景分析

可能的职业道路很多，但任何老板都只能提供一个有限的子集。

问题描述

老板提供给你的职业道路全都不适合你。

解决方法

为自己的职业生涯确定一个合理但又须付出努力的下一步。要明白这并不需要你的老板、你的职业顾问或者你的教授帮你定好。到达自己的下一步，然后继续制定整个过程直到抵达理想的目标，这都是你自己的责任。确定下一步职业目标之后，你要做的就是把中间的步骤具体化，也就是把自己前进所需采取的小步骤具体化。

注6：*http://apprenticeship.oreilly.com/wiki/show/draw_your_own_map*（需注册并登录）。

有一点至关重要，就是一定要走出第一步，即使它看起来意义没有那么大。第一步会产生一种冲力，协助你向着自己的目标前进。正是这种走出可怕第一步（以及之后的所有步骤）的意愿（即使在没有完善计划的情况下）会把你定的路线从白日梦变成现实。

不要仅仅写下上层的目标，尽量制定细小的、可达成的步骤。这些小步骤会提供一些反馈，你可以利用它们来修改自己的路线，这些小步骤还能使你能更容易地从“同道中人”那里获得帮助，从而达成自己的目标。毕竟，其他人并不能提供很多帮助来让你一下子变成Paul Graham所谓的“杰出黑客”，但他们可以帮你提供一些资源，这些资源能帮你学习Lisp、UNIX套接字编程或者达成类似的明确目标。

如果你发现自己设定的愿景跟老板为你设定的愿景不一致，而且看起来找不到协同二者差别的方法，那就考查一下其他机会，看看有没有与你所期望的方向一致的机会。记住，并非所有的学徒都要走同一条路。实际情况是：成功的学徒们所走的道路都有很多相似之处。之所以有这种相似，也并不是因为指导者死板的教导使学徒们做出了相同的决定，而是因为每一名学徒都会有意或无意地基于一组相互重叠的价值来选择生命的路线。

随着周围环境和自身价值的调整，你应该不断重新评估自己的路线。有时你的路线会跟周围的人一致，有时你的路线需要你在荒野中找出属于自己的路。某些跟我们有过交谈的学徒发现，公开自己的当前路线使他们找到了“同道中人”，同时跟现在的和过去的老板都保持了健康的关系。唯一不变的是：路线永远是自己的，你可以在任何时候重新描画它。

使用“持续动力”和“使用头衔”模式可以避免当前的头衔和薪水减少了路线图上的潜在目标。如果你需要转到一个在等级上不太重要的角色才能保持自己不脱离路线，那可以考虑“漫漫长路”模式并比较一下两者的相对重要性：是（短期）更响亮的头衔和更丰厚的薪水重要，还是一个更符合自己的目标，从长期来看能让自己飞得更高的公司重要。

Desi制定自己的路线图

我获得了一份工作，在一家创业公司中做各种各样的事情，包含数据库维护、系统管理、质量保证、源码控制，甚至一些项目管理工作。在几年的时间里一直变化角色，不久之后我开始觉得自己的两只手又想写程序了。我从SQL脚本、Perl脚本和一些shell脚本编程开始。这些脚本在我前面提及的所有任务中反复出现。我意识到，当你有时间学习自己所做的事情，而且没有实际课程的压力时，编程着实是一种乐趣。我就这样快活了一段时间，然而，老板希望我变得更加面向系统，这种来自老板的压力开始与我转向开发的期望冲突了。我失去了编程的动力，因为我的职位需要其他类型的学习和工作。我感到灰心丧气，因为这不是我想做的工作，我觉得自己已经走进死胡同了。我想放下生产运作和系统管理方面的工作，或顶多作为编写代码的附带结果来做。但公司不允许我做这样的转变。我为找到一份开发的职位费了很大周折，因为我已经离开学校四年而且没有实际的编程经验。我离开原来的公司去了另一家公司，继续从事配置管理的职位。我开始尝试把Perl引入这家新公司，结果遭遇了巨大的阻力。我意识到自己不得不再换一份工作，因为我对编写代码的向往越来越强烈了。幸运的是，ThoughtWorks决定在我身上碰碰运气。

——Desi McAdam，电子邮件

Chris突破了学习的限制

在Intrado，Dave Oberto教我学会了我所知道的SQL的一切。它是数据迁移的主管，而我是测试主管，他还是个让人惊奇的程序员。我所知道的关于C和指针算术的一切，以及所有接近机器的知识，都是Frank DeSuza教会的。我们还经常雇佣一位名叫Doug的合同工，我所知道的关于软件设计的一切都是他教会的。在那里工作我学到了大量的东西。这惹恼了周围的人。最终当别人直白地告诉我，我不再被允许继续学习，而且，虽然我非常成功地超越了职责描述的范围并打破了测试和开发的界限，但一切还是要有限度，我离开了那家公司。——我不能在那里学习编码了，所以我离开了。

——Chris McMahon，电子邮件

这些故事指出了Desi和Chris考虑问题的优先级。他们的目标是成为更好的程序员，他们不想让公司的期望或文化挡在他们达成目标的道路上。这些故事对于渴望成为开发者的系统管理员和测试专业者特别适用。太多的组织把人员分类，并采用一种目光短浅的方式对待组织的人力（或“资源”）。比起认清Desi是个渴望成为杰出程序员的人，简单地把他当成一名“系统管理员”会更容易管理。有些组织能支持人们为自己设置的大胆目标。另一些组织选择不这样做。如果你的组织是后者，那你就需要看看其他地方了，可以参考“提高带宽”和“找人指导”。

行动指南

列举三种你觉得自己在目前的工作之后可以从事的工作。然后针对这三份工作，再分别列举三种它们可能导向的工作。认真地看看所有这9种工作。这真的是你生命中接下来的几年里想做的所有工作吗？有没有遗漏的？继续扩充这张图，针对你最后增加的九种工作中的每一种再延伸出三种。这将使把图中的工作数目再增加27个。问自己，针对你的职业选项范围，以及你想把自己的职业生涯带往的地方，这份工作集合是否更具代表性？限制你选择的约束因素是什么？

如果你到现在对这张图还不满意，那就基于不同的工作重复这个练习，可能基于不同的业务或技术领域。最后再次尝试这一练习，这次试试如果放开一种你一直接受的约束条件，结果会怎样。如果你愿意搬到另一个国家，或者学习一种新的人类语言或编程语言会怎样？如果你开展自己的业务会怎样？如果那种业务只是把软件用作通向终点的手段又会怎样？可能性会比你原来想的还要多。

参考模式

“提高带宽”（第5章），“找人指导”（第4章），“同道中人”（第4章），“持续动力”（第3章），“漫漫长路”（第3章），“使用头衔”（第3章）。

使用头衔

我将把你从资深工程师提升为主管工程师，
薪水照旧，但人们会对你更加尊敬。
——Dilbert的尖发老板

情景分析

因为坚持学习，你被（正式或非正式地）聘任或提升到一个头衔中含有“资深”、“架构”或者“主管”这类词汇的职位上。

问题描述

你的职位头衔跟你在镜子里看到的自己不一样。当你在职业场合介绍自己的时候，你都会觉得必须道歉或者专门解释一下自己技能水平和职位描述之间的差异。

解决方法

不要让头衔影响了你。它只是一种娱乐，你应该在意识中将它边缘化。你可以使用头衔来评估你的组织，但不能用来评估自己。

不要被一个让人印象深刻的头衔所欺骗。妈妈或许会认为你当得起它，但让人印象深刻的头衔和责任并不表明你的学徒期已经结束。它只是用来提醒你：我们行业中技师还很缺乏。

另一种情况是：虽然你的技能水平已经超出了自己的同事，但你的头衔依旧平淡无奇。跟来自于动人头衔的夸张性一样，来自不得赏识的挫败感应该让你想到：我们的行业是有问题的。跟上面一样，你应该通过这种情形来度量你的组织以及它跟你之间的切合度，而不应该让这种挫败感拖慢了自己前进的步伐。

这一主题的另外一种提法是正式头衔与非正式头衔之间的关系。比如，虽然你正式的头衔还跟以前一样，但你在团队中已经上升到一个权威的

位置。这些非正式的头衔是难以忽略的，因为，即使它跟你的自我评估并不一致，它也会被你的同伴不断强化。这时，你跟自己的指导者和“同道中人”的关系就显得至关重要，因为他们能帮你保持立足现实的心态。

Dave看到了继续前进的信号

我写的第一个程序是一个Perl CGI脚本，在我写完它的两年之后，我的职位头衔是“资深应用开发者”。做了一番相对准确的自我评估之后，我发现自己的情况有点幽默。我并没因为这个头衔而相信自己已经达到了目标，而只把它作为自己需要继续前进、需要“自定路线”的信号。

——Dave Hoover

行动指南

为自己写出一个长长的且富有描述性的职位头衔，确定它准确地反映了你的实际工作内容和技能水平。以后不断更新这一头衔，使之保持最新，并不时想象一下：你会如何看待一个拥有同样头衔的人。

参考模式

“自定路线”（第3章），“同道中人”（第4章）。

坚守阵地

受到来自消费者至上主义者和速成社会的塞壬歌声[译注3]的诱惑，我们有时会选择一种只会带来成功表象和满意幻影的行动过程。

——George Leonard，《Mastery》（精通）

译注3：siren song，在古希腊神话中，人首鸟身的怪物塞壬用自己的歌声使得过往水手倾听失神，航船触礁沉没。

情景分析

因为坚持学习，你已经有了一定的声誉，别人认为你是个能够高效交付软件的人。在你的组织内部，对杰出工作的回报就是等级的提升。

问题描述

你获得一次提升的机会，组织想把你提升到一个不再编程的职位上。

解决方法

提升的机会将会考验你是否拥有“持续动力”，是否愿意走“漫漫长路”。大多数人把提升到管理职位等同于成功。他们觉得接受一个到管理职位的提升是不需要考虑的事情，是一个表明你走上正确道路的信号。胸怀大志的技师千万不要被蒙骗，去相信自己仍可长期做“技术管理”者。正如Pete McBreen所写：“一个人一旦停止了实践，她对技艺的精通就马上开始消退。”每一个不写程序的日子，你都是在不断远离熟练工的目标。

因此，为了让自己坚守在技艺精通的道路上，你应该跟老板一起想一想，看能否找到其他回报你的方式。这可能包括更多的薪资，或者非传统的技术领导角色，比如内部咨询工作。如果你的组织不够灵活，那你最好到别处找找机会（参见“自定路线”），而不要让自己提升到远离技艺的职位上。

“坚守阵地”是一种为软件开发“培养激情”的方法。当你接受了一次允许你全职编程的提升，别忘了“使用头衔”。

无视了周围其他人的需要，这一模式极易被运用得过于自私。然而，随着你逐渐成为更有经验的学徒，你会发现自己正在尝试改变周围的工作环境，从而使别人能坚持做自己喜爱的事情。这很容易成为一种全职的工作，除非你小心地维持平衡，或者找一些方法，为不断增加的资深程序员建立一种能自动维持的环境。

行动指南

你的老板是如何回报杰出工作的呢？如果他们目前采用的回报形式不那么吸引人，开始考虑一些可以让他们用来回报你的其他方法。考虑是否存在可以为你稍作松动的制度约束。或许在你的合同中有一些限制条款，或者你有一种激进的想法需要支持。准备一份具有三种不同回报方式的列表，这样当拒绝那个提升机会时，你已处在一个可以协商的位置上，而且是基于你很清楚自己想做什么这一事实。

参考模式

“自定路线”（第3章），“培养激情”（第3章），“持续动力”（第3章），“漫漫长路”（第3章），“使用头衔”（第3章）。

另辟蹊径

不要因为他们没有走在你所走的路上，就认为他们已经迷失了。

——H. Jackson Brown Jr.，《Life's Little Instruction Book》
（人生的小小忠告）

情景分析

你采用了“自定路线”模式，而且勤勤恳恳地遵循着。

问题描述

你所描绘的路线使你远离了那条“漫漫长路”。

解决方法

沿着自己定好的路线走，记住自己在学徒期学到的东西。

你已经在“漫漫长路”上走了一些时候了。但因为“自定路线”，你现在认识到这条路不再是适合你的选择。你找到了另外一条路，其回报与你现在的价值更合拍：更多时间与家人呆在一起，更多的薪水，或者一

种新的职业引起了你的注意。不管是什么，都意味着跟你的技艺和“漫漫长路”说再见。可能是永远，也可能不是。

即使永远地离开原来的道路，在这一路上你所形成的价值和原则也将一直陪伴着你。正如Dave在他不再做家庭问题顾问时所发觉的，他不能做Prospero[译注4]那样的选择（书烧掉，工具打碎），相反，他把那份职业中获得的经验教训带到了新的工艺职业中。同样的道理也适合你。

从Ade在ThoughtWorks那时起，Ivan Moore就是他的指导人。当我们访问Ivan时，他讲述了在第一份IT职业之后，他是如何到了一个希腊的岛上，在那里呆了六个月并成为一名帆板运动的教练。他发现自己喜欢教别人帆板，但这并不完全让他满意，因为他再也不需要脑力劳动了。之后，他费了好大的劲才回到了原来的行业，因为“多数大公司的HR人员都不喜欢他的经历。”

我们有一些同事离开软件开发并成为教师、帆板运动教练或者全职的爸爸妈妈。我们尊重他们的选择。如果他们愿意回来，我们会用张开的双臂欢迎他们，因为我们相信：那些经历会带给他们一些可以分享的新视角。遗憾的是，传统的软件组织不一定会如此欢迎。他们常常把这些弯路看作职业生涯中的缺口，因此你必须为此做正当的解释。他们会期望你能给出一个符合他们价值系统的合理解释，说明白当初为什么离开了，现在为什么又回来了。

尽管有这样的风险，你仍然不必害怕在自己的一生中做点不同的事情。如果你离开了软件开发，你会发现不管自己去哪里，像严谨的思考以及将涉及大批量数据的任务自动化这样的习惯仍将对你有用。不管你选择什么样的未来，你过去做软件技师的经历都能使之更加丰富。

Larry绕道去做家庭问题专家

在某种程度上，我可以远离计算机，却不能远离跟人有关的问题。1976年7月，当我跟计算机领域说再见的时候，我想我已经解脱了，我甚至像美国庆祝独立二百周年那样宣告了自己的独立。我花了不止十年的时

译注4：普洛斯彼罗，莎士比亚剧作《暴风雨》中的人物。

间在私人诊所和事务机构中工作，接受家庭问题医生的相关训练，处理夫妻、家庭和问题青少年身上遇到的麻烦。然而，来自世间万物的力量协同作用，最终还是把我引回了科技前沿。

——Larry Constantine，《The Peopleware Papers》
（人件集——人性化的软件开发）

行动指南

如果由于某种原因，你不能继续做一名软件开发者，你会怎么办呢？写下一些你认为自己会乐于从事的其他工作。找一些正在做那种工作而且热爱它的人。问他们所爱的是什么，并将它们与你在软件开发中所喜爱的东西比较一下。

参考模式

“自定路线”（第3章），“漫漫长路”（第3章）。

总结

对学徒的“漫漫长路”旅程提供直接支持的模式可以按照无数种方式来组合。它们在本章中的次序并不表明任何线程的前进序列。比如……

你可能相对从容地走在“漫漫长路”上。工作环境并不让你感到沉闷，你对技艺有强烈的激情。你很优秀。你被提升到组织称之为“架构师”的职位上，并且仍然全职编程。你的“准确自我评估”表明：是该运用“使用头衔”模式来评估一下组织的水平了。这足以让你坚守在“漫漫长路”上，你可能并不需要其他模式。

……或者……

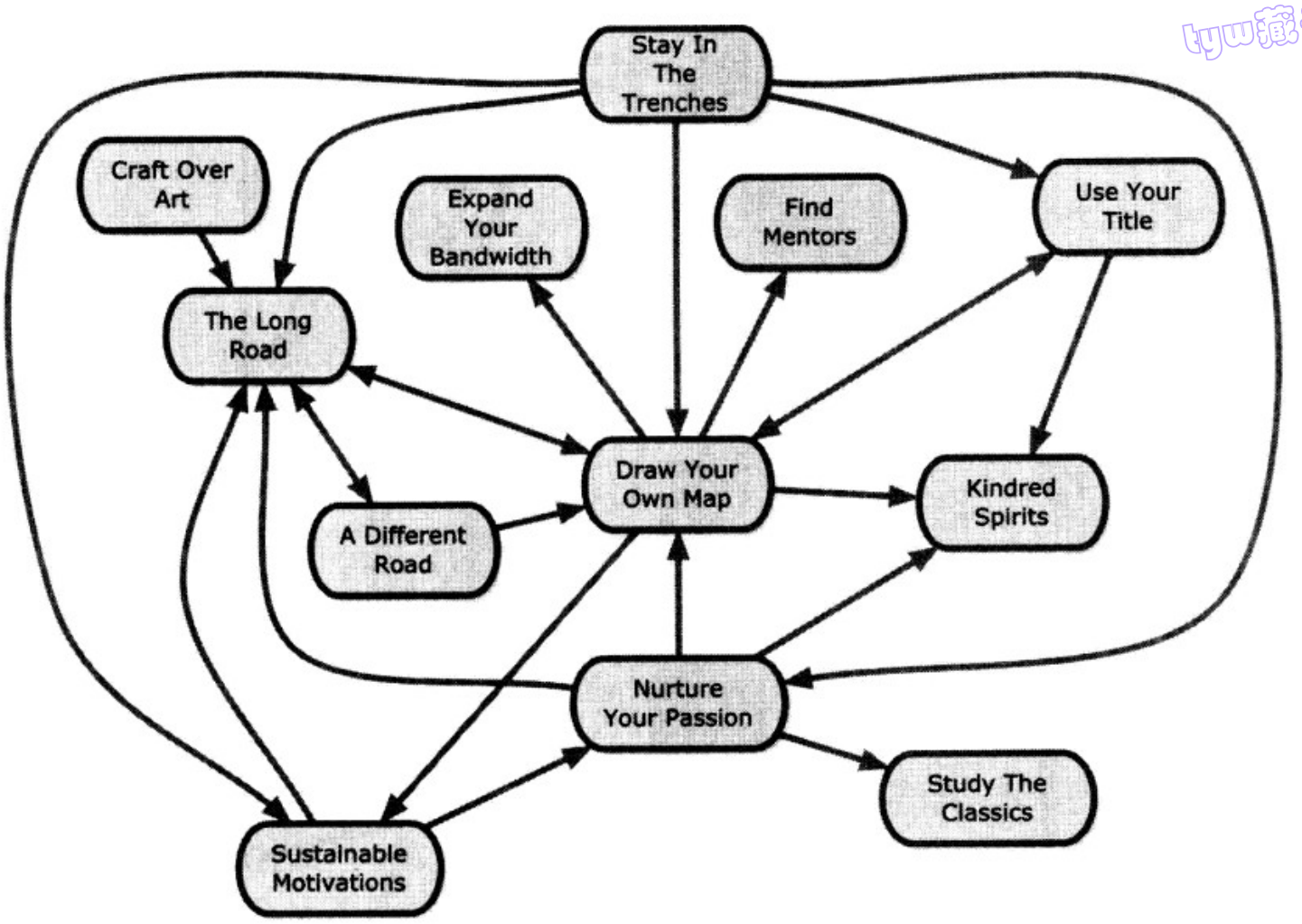
你的组织强调的是薪酬回报。组织内部有一股恒定的但没有人说出来的暗流，它迫使人们关注于赚取更多的薪水；毕竟，赚更多的钱意味着你对公司更有价值。你意识到了这股暗流以及它可能对你带来的危害。你“培养激情”并致力于维持自己的“持续动力”。你对“技重于艺”的关注使你提高了自己的声望，组织给了你一次提升到项目经理的机会。

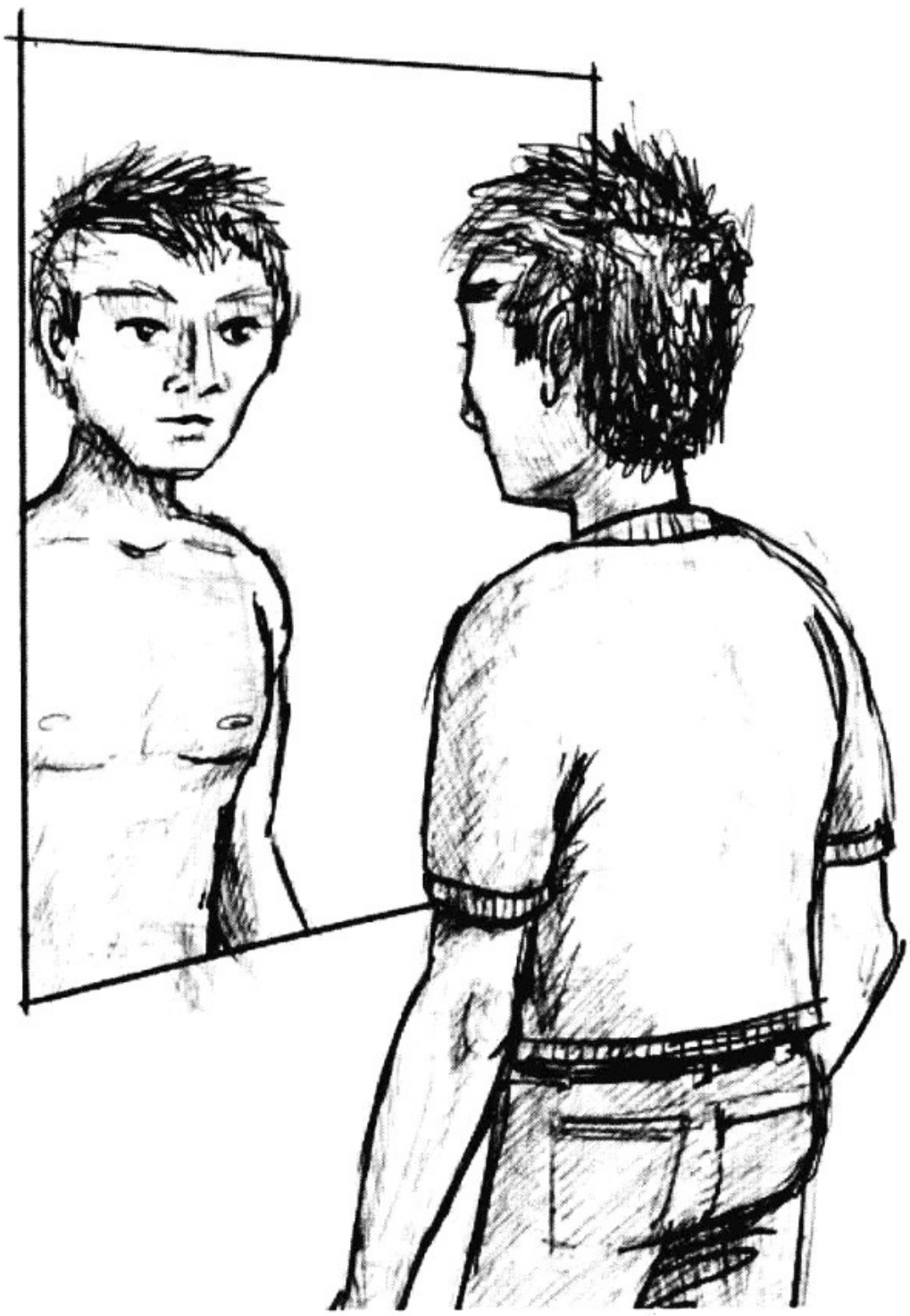
但通过“坚定阵地”和“自定路线”，你和老板一起定义了一条让自己坚守在“漫漫长路”上的职业道路。

……或者……

你从小就开始编程。你写程序是因为你乐于创造漂亮的、优雅的解决方案。但现在，在公司的环境中，你面对的是不令人愉快的任务，还有要求你交付功能但并不关心实现是否优雅的客户。你意识到：在可见的未来，你编程的动力无法向“漫漫长路”看齐。你采取措施来“培养激情”，通过关注“技重于艺”来培养更多的“持续动力”。时间长了，你逐渐有了与客户建立牢固关系的动力。

你可以看到，组合这些模式的可能性有无数种，就像学徒们所处的环境一样。


Stay In The Trenches
Craft Over Art
Expand Your Bandwidth
Find Mentors
Use Your Title
The Long Road
Draw Your Own Map
A Different Road
Kindred Spirits
Nurture Your Passion
Study The Classics
Sustainable Motivations



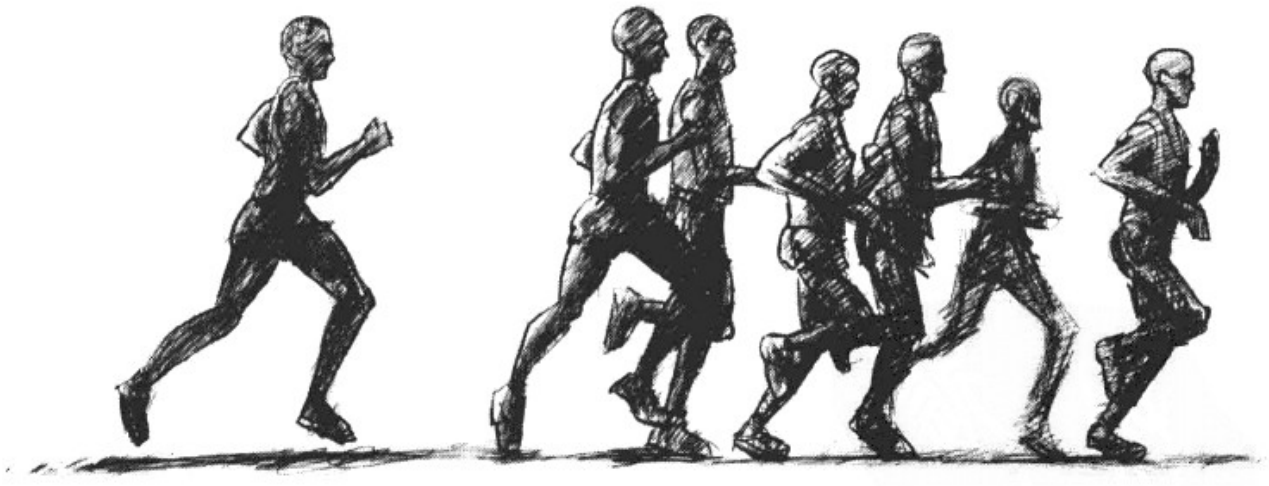
第4章

准确的自我评估

快速学习的人们面对的主要风险之一就是变成小池里的大鱼。尽管小池塘和大鱼本身都没有错，但有必要让大鱼知道在全球的池塘网络中还有其他的池塘，这非常关键。更重要的是：大鱼还要知道巨型鱼的存在，那些体积甚至超过池塘大小的大鱼。

聪颖而且努力的学徒千万不要对自己的成功自鸣得意。在软件开发领域，超越平庸是很容易的，因为有太多的人只超出平均曲线一点点就开始满足了。你必须去追寻那些技艺精湛、学徒甚至无法想象其水平的其他团队、组织、熟练工和师傅，向他们学习，以此来抗争趋于平庸的倾向。

你必须愿意放下自己已经感知到的能力，意识到自己在“漫漫长路”上只走了一小段。你的目标并不是变得优于“平均水平的开发者”。你的目标是度量自己的能力，并找到比昨天的自己做得更好的方法。我们都走在同样的旅途中，将自己跟其他人比较是有益的，前提是要帮我们找到互相帮助、共同提高的方法。



只求最差

宁为狮尾，不为狐头！

——Tractate Avot

情景分析

你已经“释放激情”，并利用所有可以抓住的机会来学习新技能。结果，你的成长超出了团队的需要，甚至可能超出了整个开发组织的需要。

问题描述

你的学习速率曲线已经变平。

解决方法

让周围多些水平比你更高的开发者。找一个更强大的团队，在那里让自己成为最弱的成员并拥有成长的空间。

“只求最差”是这一模式语言中的种子模式。它是从Pat Metheny提供给年轻音乐家们的一些建议中提出的。Pat Metheny说：“在任何一支乐队中，都要做最差的。”[注1]Pat的建议触动了Dave的一根心弦，成为他开始撰写本书的原因之一。

Dave找到一个更好的团队

我的第一份编程工作是在一家.com的创业公司中度过的。工作五个月之后，就迎来了互联网炸弹（dotbomb）泡沫破裂。烟消云散之后，我降落在一家非营利组织的IT部门。那是一个极好的避风港湾，我在那里平安度过了几年的行业低迷期，但它与我前一家雇主比起来，开发的节奏实在太慢了。两年后我已经学到了很多东西，但工作没有任何挑战，也看不到组织的状况有任何改善的希望。充当团队的技术架构师起初为我提供了极好的学习体验，但我觉得自己充当架构师的角色是荒唐的，这促使我开始到别处寻找工作。我找工作的唯一目的就是提高自己的学习速度，就我所知，能达到这一目标的最好方法就是让自己周围有一群杰出的开发者。非常幸运，一年后我加入了一支团队，里面有多名世界一流的开发者。那是一次不可思议的挑战，但也是一次宝贵的机会。

——Dave Hoover

注1：Chris Morris的博客，“Be The Worst”（只求最差），参见：*http://clabs.org/blogki/index.cgi?page=/TheArts/BeTheWorst*。

处在一个强大的团队中能使你“感觉”自己工作得更好。团队的其他成员经常阻止你犯错误，并帮你平稳地从错误中恢复正确的结果，以至于虽然学到的东西没有自己想象的那么多，你却意识不到这一点。只有在独自工作的时候，你才会发现团队在多大程度上提高了你的生产率，并意识到自己学到的东西有多丰富。对团队中那些水平最低的人来说，这也使得“且行且思”和构造“质脆玩具”特别重要。这两种模式都提供了一些机会，让你可以在团队环境中后退一步，与更有经验的队友“密切交往”，在此过程中看看自己养成了哪些习惯、学习了哪些技术和知识。

前面的图说明：作为团队中最弱的一员，你应该比其他人更用功。这是因为你的目标不是保持“最弱”，而是从最后面开始一路赶上去。要做到这点，你需要不断找到改善的方法，不断模仿更强的开发者，直到你跟团队其他成员处在同一个水平上。没有这种集中精力向团队学习的意识，你会面临多种风险。

首先，你面临拖累团队速度的风险。其次，由于好的团队不会（长期）容忍某个成员只是来做一回过客，如果你远远落后，以至无法赶上，或者看起来无法足够快地赶上，那你还面临炒鱿鱼的风险。加入强大团队还有另一种副作用：最终你会对你自己以及你的技能水平产生不好的感觉，除非你积极地磨炼技能。最好的情况下，这会促使你进步。但跟所有“要么游起来，要么沉下去”的策略一样，当失败时你会发现自己已经沉溺。这也解释了为什么“建立馈路”从而遇到麻烦时让自己知道非常重要。这种反馈能告诉你团队是否已超出你太远，或者团队是否对努力一路追赶的人们并不友好。

跟“坚守阵地”一样，“只求最差”与鼓励人们尽快获取更高级职位的常规文化相悖。但作为学徒，你应更重视学习技能的机会，而不是扩大并维持自己的权威。有时，这意味着虽然你正领导着一个团队（参见“深水区域”），但作为学徒，你通常更应期望被别人领导。

故意作为最差的一员加入一支团队，这有自私的一面。为抵消这一缺点，可以用“打扫地面”和“具体技能”模式来补充“只求最差”。

“打扫地面”意味着：为直接给项目增添价值，你特意去做些打下手的任务。开发“具体技能”会增加你对软件开发过程的贡献，而且这是学徒这一角色的根本。没有这类贡献，该模式会导致强大团队被严重削弱——正如Jamie Zawinski在写给Mozilla项目的那封不怎么光彩的公开辞职信[注2]中所指出的。最终，你对团队快速增长的贡献会成为这些强大团队愿意冒险拉你入伙的理由。

Jake Scruggs在谈及他在Object Mentor的夏令实习期时说得很好：

> 毫无疑问，在Object Mentor工作，最酷的事情就是只需探一下身子就可以去请教David、Micah、Paul、James或者……看，坐在我旁边的每个人对编程都持有高见，跟他们的卓越才华一样酷的是：与优秀的程序员一起工作是一种更好的学习方法。[注3]

除了在“找人指导”方面给你提供帮助，与优秀的开发者合作还能帮你维持更加准确的自我评估。然而，对那些准备成为熟练工的更有经验的开发者来说，这一模式并不适当。在那一阶段，你应尝试去指导新手，把别人曾经给你的机会再给予别人。

Brian的选择

加入Obtiva把我放回了食物链的最底端。现在，我是个绿色的“软件学徒”，在我领导一个团队之前，这会持续相当长的一段时间。我将从事学习，而不是带队。我被降级了。

为什么会有人选择这种事情？

首先，到一个杰出开发者的团队中做一名最弱的队员，这种机会是环境、设备和金钱所无法取代的。过来之人知道如何避免前路上的陷阱，在这些人身边学习的机会是无可比拟的。跟优秀的开发者结对，这种机

注2：Resignation and postmortem（辞职信和事后检讨）：*http://www.jwz.org/gruntle/nomo.html*。

注3：Jake Scruggs，“My Apprenticeship at Object Mentor”（我在Object Mentor的实习期），参见：*http://www.jikity.com/Blah/apprentice.htm*。

会更是非常宝贵的。如果你对编程还很陌生，还不曾有过结对的机会，那你需要去寻求这样的体验机会。[注4]

行动指南

列举你知道的所有团队，包括开源项目、其他部门和其他公司的团队。按技能水平将这些团队排个顺序，然后从中找到一支对寻求上进的新人敞开大门的团队。这可能需要你加入多个邮件列表，并向各种不同的人问问题，借此度量他们的相对技能水准。这一过程结束时，你将更善于比较技能水平，甚至有了新的团队！

参考模式

“质脆玩具”（第5章），“具体技能”（第2章），“建立馈路”（第5章），“找人指导”（第4章），“且行且思”（第5章），“密切交往”（第4章），“坚守阵地”（第3章），“打扫地面”（第4章），“深水区域”（第2章），“释放激情”（第2章）。

找人指导

对初学者来说，不论他是从培训班开始还是通过自学，
走上软件技能之路的第一步都是找一名技师来带他。

——Pete McBreen，《Software Craftsmanship》（软件工艺），第96页

情景分析

你意识到自己并不是第一个走这条长路的人，你花了大量的时间在探索死胡同。

问题描述

走在一条路上，你不知道下一个拐角处是怎样的，也不知道如何为之准备。你需要帮助和指导。

注4：Brian Tatnall，“New beginnings with Obtiva”（在Obtiva的新开端），参见：*http://syntatic.wordpress.com/2007/05/18/new-beginnings-with-obtiva/*。

解决方法

找那些走在你前头的人，努力向他们学习。

理想的情况下你将找到一位师傅级的技师，而她愿意收你做学徒。在整个学徒期你将处在她的监管之下，基于师傅的名望来构建自己的未来。然而，在今天的世界上，这种理想情况格外罕见。

我们的领域非常年轻，因此公认的师傅很少。此外，作为学徒，你很难判断到底谁是真正的师傅。因此，更有可能的情况是：你的学徒期被连续的多位指导者监管，其技能的精通度各不相同。

真正的学徒必须融入师傅的生活中，摸爬滚打，重视每一次被关注的机会，特别是跟师傅面对面学习的机会，最好是肩并肩地合作。话虽如此，但有可能你根本接触不到最有影响力并且对你最有帮助的指导者。他们可能生活在一个不同的国家，甚至在很久之前就去世了，比如像Edgar Dijkstra这样的人。但这并不影响他们做你的灯塔，为你照亮前进的路。

> 如果你最终发现找到的老师不适合自己，那首先应该想想自身的问题。可能你的期望值过高，没有哪个老师能达到。
>
> ——George Leonard，《Mastery》（精通），第71页

在尝试“找人指导”时，学徒必须记住：我们都走在“漫漫长路”上，没人知道全部。你很容易觉得自己的指导者肯定是位师傅，因为她知道的东西比你多太多了。你必须抵制这种想法，否则可能因为指导者不可避免的弱点和盲区而幻灭了自己的希望，觉得自己不可能从她身上学到什么了，虽然实际上她还能教你很多。

Dave指到了指导者

到2002年夏天，我已经有了近两年的编程经历，并开始感觉到初学者和经验极其丰富的从业者之间的巨大差别。那年夏天我阅读了Pete McBreen写的《Software Craftsmanship》，这促使我去寻找指导者。这本书让我明白了：如果想成为优秀的开发者，我将不得不去接触更

> 有经验的开发者，并调整自己的学徒期。我已经进入了“提高带宽”的状态，而且刚刚参加了ChAD，即芝加哥敏捷开发者（Chicago Agile Developers）用户组。就是在那里我（不同凡响地）把自己介绍给了用户组的组织者Wyatt Sutherland。读完Pete的书之后，我马上给Wyatt发了封邮件，让他知道我很想有人指导一下。这是一封让人感到不自在的邮件，但却获得了巨大的回报。Wyatt回信建议我们定期在吃早餐时碰个头，讨论一下我们正在做的事情。到第二年，Wyatt成了我非常重要的指导者。虽然我们从未“密切交往”，但跟一个像Wyatt（一个很受尊敬的软件顾问和世界级的大提琴演奏家）这样的人建立关系，对我这个未经训练的程序员新手来说，这是一次巨大的信心提升。Wyatt的指导对我在敏捷软件开发方面的进步起到了关键作用，并且鼓励了我，让我相信自己有能力加入像ThoughtWorks这样的软件开发组织。

找人指导自己，从概念上很容易理解，对学徒期也格外重要，但它还是会很困难。诚然，找到一位作者、技术会议发言者、流行开源项目的代码提交者或者成功网站的开发者并不难。但困难来自两方面。首先，这些人可能没兴趣指导别人；然后，跑出去做一件请求别人“指导学徒”这么奇怪的事情会让自己有说不出的胆怯。这很像那些与“深水区域”有关的风险。要记住，被一位潜在的指导者拒绝或者认为你奇怪的概率并不高，而潜在的回报却是巨大的。即使那人没兴趣收你为全职学徒，邀请她出来共进午餐也会是很有价值的时间和金钱投资。如果你对于精通技艺是严肃的，那就勇往直前地找人指导吧。技术拔尖的开发者都是这么过来的，很少会有记不起指导者曾对他产生过的巨大影响的。

学徒期的训练不会孤立地发生，有人会走在你前面，同样也会有技能水平尚不如你的学徒。寻找指导者是一方面，另一方面你也必须向那些需要你指导的人提供帮助。将自己从指导者那里学到的东西传递下去，要完成到熟练工的状态转变，这是你可以依赖的方法之一。

行动指南

挑一个拥有活跃邮件列表的工具、库或者社区。订阅这个列表，但开始不要发布任何消息。潜水即可。过段时间你将开始理解这个社区的价值，并了解哪些订阅者是耐心的老师。有了这些了解之后，到下次技术

会议时，你就可以去寻找这一列表中的成员，看他们是否有兴趣为你所学的功课提供一些非正式的建议。

参考模式

“深水区域”（第2章），“漫漫长路”（第3章）。

同道中人

最强大的东西就是由致力于解决
相关问题的能人组成的社区。
——Paul Graham，《Hackers & Painters》（黑客和画家）

情景分析

你的学徒期已经过了几个月或者几年，而开发组织的文化却让你失去了信心。

问题描述

鼓励软件技能的组织文化非常罕见。你发现自己无人指导，束手无策，而且周围的气氛看起来与你的志向并不一致。

解决方法

为保持前进的势头，特别是当找不到全职指导者的时候，你需要找到那些所走的道路与你相似的人，跟他们保持频繁联系。因此，你应该去找一些像你一样希望自己优秀的人。

“漫漫长路”并不是单个人孤独行走的道路，特别在学徒期的那几年中，你需要同志和友情。这一模式原理上很简单，对某些人（我们的外向型同伙）来说实践起来也不难。然而，其他一些人实践起来可能有困难。有些关系是短暂的但会随着职业而变化；有些关系是持久的且能帮你“培养激情”。以下几则故事中的例子能证明“同道中人”的力量。

- Dave在2002年阅读了《Extreme Programming Explained》（解析极限编程——拥抱变化）一书，然后一头扎进了XP和敏捷社区中。他自筹路费去参加2002年XP/敏捷全球大会（XP/Agile Universe），那一年是在芝加哥近郊举办的，去一趟很方便。在大会上，Dave遇到了Roman，之前已经有人通过某个本地用户组的邮件列表在网上向Roman介绍了Dave。Dave和Roman相约午餐时一起讨论Joshua Kerievsky正在写作的《Refactoring to Patterns》（影印本《从重构到模式》，机械工业出版社引进），Roman在一家大型跨国银行工作，而Dave在一家老式臃肿的非营利组织中任职；这并不让人惊讶，他们都喜欢从各自开发组织内平淡无奇的工作中解脱出来，连续几年他们每周都见一次面。他们并没有一直讨论Joshua的书（因为Dave首先需要读一读《Refactoring》（重构：改善既有代码的设计）和《Design Patterns》（中译本《设计模式：可复用面向对象软件的基础》，机械工业出版社）），而是花时间在各种方面：讨论其他的书，像《Peopleware》（人件），在Dave的笔记本上学习Ruby，分享恐怖故事，并互相为他们那几年中遇到的各种问题思考解决方案。
- Steve Tooke向我们讲述了2004年遇到Shane的故事，那时他在英国曼彻斯特的一家公司工作。Steve是个热情洋溢的年轻程序员，Shane是在新西兰总部任职的富有经验的程序员。虽然相隔万里，他们的交流却改变了Steve的职业。Shane把诸如《Design Patterns》这样的书籍介绍给Steve，这使他们有了一种描述面向对象设计的共通语言，那时他们都在搞面向对象设计。相隔多个时区的沟通障碍意味着Shane并不能提供很多直接指导，但对Steve来说，这种联系本身，以及知道开发组织中有另外一位专注于卓越的家伙足以对他的工作产生了重大的影响。

指导者是那些你希望仿效他们的人，因此可能会让你觉得有点距离感，有时还让人敬畏。但另一方面，社区为你提供了安全的研究和学习的环境。或许你对高级的JavaScript感兴趣，而你的“同道中人”之一正在研究Haskell。你们可以随意地互相展示各自学到的东西，谁也不必受

谁领导。这跟基于指导者的关系不一样，在那种关系中，学徒会觉得有义务放下自己对JavaScript的兴趣从事Haskell的研究，原因仅仅是他的指导者认为那是更好的语言。记住这一点，然后通过一个“同道中人”组成的社区来补充“找人指导”模式，这个社区中都是些让你交流起来没有心理负担的人。

尽管由想法类似的人组成的社区能带来很多好处，你还必须了解集体思维。要让自己一直有能力问一些使整个社区吃惊的问题。运用这点智力上的距离来产生令人尊重的不同意见，它会使你的社区保持健康。社区的健康度可以通过它对新思想的反应来衡量。社区会在激烈争论和实验之后拥抱那种思想吗？还是直接拒绝那种思想以及那个提出它的人？今天持不同意见的人可能是明天引领新思潮的人，有些人相信因循守旧是加入社区的代价，而你能提供给社区的最有价值的服务就是保护它不要被这些人支配。

行动指南

基于自己使用的工具、了解的语言、一起共事过的人、读过的博客，以及让你受过启发的思想，列举一下所有可以加入的社区。确认其中有哪些能在你的城市搞线下的实际聚会。逐一参加所有这些聚会，确定哪一组人最有趣。

如果所有用户组都不会在你附近定期聚会怎么办？遇到这种情况，那你就获得了一次极好的机会：自己创办一个这样的集会。在你的地区发起一个软件技师的定期聚会。这个事情不会像你想象的那么麻烦的。只是不要过早地限制了成员和主题的范围。相反，到这一地区开发者可能看到的地方，尽情地散发广告就行了。

随着组织的壮大，尽情去探究更宽广、更奇特的主题领域，直到你拥有一个由不定期参加聚会的成员组成的核心小组。久而久之，因为这个自发产生的核心小组的存在，用户组的风格自然会产生。没必要每隔两星期都是同一批人参加——这正是说他们“不定期”的原因。像“极限编程周二俱乐部”（Extreme Tuesday Club）这样的用户组拥有好几百名

“会员”；而每个星期二只会有大约十几个人来参加。如果你的用户组变得足够大足够有活力，即使你不在那里的时候它也会维持下来。这时你就可以说自己拥有一个社区了。

参考模式

“找人指导”（第4章），“培养激情”（第3章），“漫漫长路”（第3章）。

密切交往

我喜欢有足够的自由度，这样才能理解或者想出一些东西，
但我也喜欢合作。我会寻找那些能让我从共事的人
那里学到东西的项目，并会因此而更有活力。
——William Kempe

情景分析

尽管你有定期见面的指导者和同道中人，当回到软件开发的时候，你仍然是独自一人。

问题描述

你的生产率进入了稳定状态，你的学习停滞了，你感觉有更高级的技术和方法而自己却抓不住。

解决方法

找机会跟另外一个软件开发者坐在一起，肩并肩完成一项实际动手的任务。有些东西，只有当你跟另外一名开发者坐在一起完成共同目标的时候，你才能学会它。

这一模式可以密切联系“同道中人”。Dave就是这样的，他与Roman结伴，之后便通过午饭时间密切交往，一起学习像Ruby编程和Eclipse

插件开发这样的技术。然而，即使Roman当初没有成为Dave的同道中人，但只要跟一个天才的程序员并肩工作，哪怕是做宠物项目，也同样能使Dave获益良多。总会有一些细微的技艺，只有跟一个同事近距离合作才能学到。这些技艺通常被认为太琐碎，没人会在教别人的时候提到它，但它们的影响却可以累加起来。作为一名开发者，与Roman的合作对Dave所取得的进步至关重要，因为那时在他的职业生涯中，很少能有机会跟天才的开发者一起工作。

“结对编程”这一实践就是该模式的一个例子，学徒们应该寻找机会，到一个使用这种技法的团队工作。对学习来说，结对编程是一种非常好的技艺，但它也是一种复杂的活动，而且它本身不见得总能让人觉得好。但是，如果你有效地使用它，它就会成为最强大的学习方式之一，特别在你向指导者学习的时候。可是你如何知道结对编程是否被有效使用呢？学徒们又能做些什么呢？

当你通过结对编程来实施这一模式的时候，你常会感觉自己走丢了，或者落后于同伴太多。这不代表结对编程失败了——它只能说明你需要问一些问题，让过程慢一点，或者忍受这种迷失的感觉并努力拣起那些你能理解的片段。但是，如果你连续几周一直觉得自己落后，觉得自己开始绝望了，那就是做点改变的时候了。你可能卡在一个不太好的结对搭档身上，也可能你需要给他一些建议来改善你的体验。作为学徒，你或许没有太多力量来改变自己的处境，但如果工作在一个多人的项目中，你就有机会每天或每周不断地轮换结对搭档。这种轮换能让你慢慢走出困境，重新回到进步的道路上。此外，如果你正在实践测试驱动开发，你可以建议大家使用“乒乓编程”[注5]，并把它作为提高自己参与度的方法。

根据Richard Sennett的《The Craftsman》（技师）一书，最理想的技师工场是这样一种地方：你在那里可以“吸收那些只可意会、不可言传的知识；靠每天的点滴进步积累成一种实践习惯。”（第77页）。由于如此理想的环境如今非常少见，我们不得不使用“密切交往”模式作为现

注5：Dave Hoover，“Ping-Pong Programming”（乒乓编程），参见：*http://www.stickyminds.com/s.asp?F=S9101_COL_2*。

代的替代品。这一模式的应用不只限于结对编程。我们的目标是接触到那些技能更加娴熟的人，学习他们的日常工作习惯，观察他们靠什么方法将那些习惯逐渐磨炼成高超的技能。这些习惯不只限于编码，也可以延伸到软件开发的方方面面。

例如，你可能在一篇学术论文上，一份演讲稿上或者开源项目的一次冲刺（sprint）中跟某人合作。或者，就像Ade当初那样，你主动去帮助一个人，这个人用UNIX shell脚本写了一套内容管理系统，在设计其中的依赖管理模块时希望应用图论知识。有些人想用非底层的工具去解决一个你可以用高级语言自动解决的问题（或者相反），跟他们共享一块白板，你就能迫使自己暂时按他的思路来思考，从而更有效地交流。即使你最终拒绝了他的观点，也会获得一种新的看待问题的方法。这一视角可能是未来某个问题的正确解决方案，即使眼下你觉得它很不靠谱。

不管你对密切交往的体验是正面的还是负面的，你都应该“记录所学”，以后可以回过头来想想自己的经历。有一天你很可能处在你的结对搭档当初所处的同一位置上，这时，对于坐在你旁边的新手，以前的经验会让你对他的心态有非常宝贵的感悟。

行动指南

找来一位你所认识的，对启动或加入一个开源项目表现出浓厚兴趣的人。每周安排一个晚上，跟他一起做这个项目。看看你们会在多长时间内相互支持彼此。忙碌生活带来的压力将不可避免地削弱你们合作背后的动力；如果出现这种情况，你们必须适应它，并设法保证项目一直往前走，直到你们找回动力。当然，如果动力再也没有找回来，那就看你自己了，可以去组建新的、能让你学到新东西的伙伴关系。

参考模式

“同道中人”（第4章），“记录所学”（第5章）。

打扫地面

在传统工艺中，新人刚开始时做师傅的学徒。
他们从相对简单的任务开始，然后随着学习和技能的不断熟练，
慢慢转向更艰巨，更复杂的任务。
——Pete McBreen，《Software Craftsmanship》

情景分析

你是某个项目中的新学徒。

问题描述

你不确定自己在团队中的位置，团队也不确定你的水平。你想找一种方法为团队出力，赢得团队信任，并提高自己作为技师的声望。

解决方法

主动去完成简单无趣却又必须完成的任务。这是一种可以尽早为团队的成功付出努力的方法，因为它表明你能完成高质量的工作，即使这种工作看起来无关紧要。当然，结果可能证明项目中看似单调无趣的部分实际极其重要，所以在任何项目中敷衍这一部分的质量都会导致日后的麻烦。

Paul真的扫地去了

我曾经在一个正式的软件学徒环境中接受培训。在我17岁时Object Mentor就雇用了我，大学时每个暑假和寒假我都回到那里。学徒期开始时，我并不知道如何编写软件。我写了一些代码来创建简单的程序和脚本，纯粹为了好玩。正式做软件学徒时，我几乎找不出任何可以为公司的业务带来价值的地方。我不会写软件，当然也不会教其他人写软件。

那时，Object Mentor雇用了许多成功的极限编程（eXtreme Programming，XP）领域的领军人物来讲授XP课程。作为迷恋明星的年轻学徒，我被任命去编制他们所设计的完美的结对编程表格。此外，我

> 还要把电脑集中在一起，为我们的练习安装所需的软件，并打扫房间。这些任务使我很自信地成为团队的一部分、公司的一部分。在竭尽全力学习编码的过程中，我还能通过一些具体的才干和任务为团队出力，而且并不需要很高的技能。
>
> 随着学徒期的进展，慢慢地我可以承担技术上更有挑战性的任务了，但常常还是那些分派给初级开发者做的事情。一些像搬动服务器，设置备份系统，还有更新网站内容这样的工作。这些任务能帮我获得一些小胜利，那时我还很难从编写代码中获得这种胜利。
>
> 充当传统的学徒角色也培养我了谦虚和尊重高级技师的习惯。我记得有一次Bob Martin大叔来到我们的房间，看到垃圾桶已经溢出了，于是换了垃圾袋。我的指导者批评了我，恰如其分指出：倒垃圾并不是师傅的工作。虔敬是我需要学习的重要一课。
>
> ——Paul Pagel，电子邮件

和Paul不一样，大多数学徒不需要真的去扫地。但是，你愿意主动承担的任务对团队的健康成长同样是很重要的。

这类任务包括维护构建系统，产品支持，响应维护需求，bug修正，代码复查，消除技术债务（technical debt），搭建项目wiki，更新文档，为其他人的想法充当传声筒，等等。通常，你会关注风险更低的系统边缘部分，而不是常常带有大量依赖性和极高复杂度的核心。Jean Lave和Etienne Wenger观察了不同行业中的学徒，发现“在工作流程中，新手的任务往往被置于分支的末尾，而不是一连串工作片段的中间”（《Situated Learning》（情景化学习），第110页）。这类边缘任务会使团队受益，对作为学徒的你也有好处，因为这些杂务在学校的课程中常常被略过了，做一做能填补知识中的漏洞。当你成为熟练工时，这种体验还是能带来帮助，因为很多带你的师傅都明白：有个人去做单调的工作是多么重要。毕竟，如果没人打扫地面，那么富有魅力的工作也无法做出来，因为团队已深陷脏乱之中。

当然，花费了大量时间和金钱去接受计算机科学教育之后，你可能难以接受“打扫地面”的工作。理论上，你已经为自己应得的东西付出过

了，你曾频繁地通宵调试，也已经忍受过来自教授的无数奴仆任务。不幸的是，在工作岗位上，你受到的教育并不如你认为的那么有价值。诚然，很多组织招人时把计算机科学的学历作为高优先级来考虑，但被人雇用和加入团队是两码事。一旦你入了行，那些教育背景只能抬高人们对你的期望，期望你第一天就能交付更好的结果（但愿那些教育背景能为你的第一天多准备一些东西吧！）。如果你是自学成才，在之前的项目上已经付出过，情况也是一样。不论你来自哪里，在加入一个新项目时都是从头开始。你应该借此机会发出你想为团队效力的信息，即使那意味着承担一些不怎么诱人的工作。

运用这一模式会导致一些负面结果。其中之一是你最终成了团队中的地鼠，被要求永远去做其他人都不愿做的奴仆任务。而且，由于你选择的工作单调乏味，别人不太会认为你已经跨越了自己的边界，这样你就会面对另一种风险：无法将前期的成功转变成请求更有挑战性任务的理由。另一种负面结果是：你发现自己害怕去做除“打扫地面”之外的任何事情。还有一种风险，如果只做琐碎的任务而没有更高的前后一致性，你会无法理解更大的图景。如果你发现自己处在这样的情形中，那就努力去“培养激情”、“释放激情”，为自己打打广告，并寻找各种机会证明自己能做更高级的工作吧。

行动指南

被你的团队拖延了几个月的最邋遢的任务是什么？它应该是所有人都抱怨，却没有人愿意解决的那个。你去解决它。不要捂着鼻子强迫自己去做；看自己能否找到一种超出人们预期并为自己带来乐趣的方式，创造性地解决这个问题。

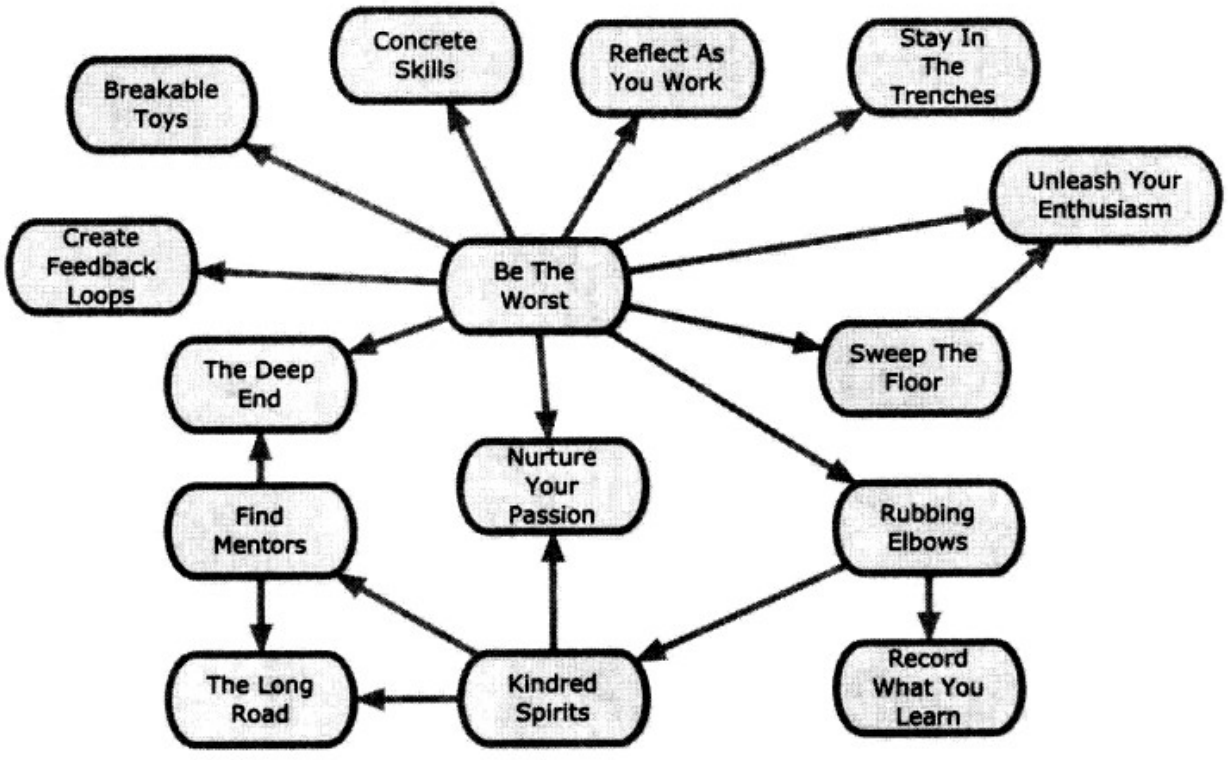
参考模式

“培养激情”（第3章），“释放激情”（第2章）。

总结

谦虚是成功学徒过程的基础之一。与自己的志向结合，谦虚能让你集

中精力，并保持沿正确的方向前进。没有谦虚，你很容易过早地宣告自己学徒期结束，并遗漏一些重要的课程。可能你对自己交付的一个重要项目和子系统感到自豪，并相信那足以证明你已成为熟练工。或许如此吧。但你在多种平台上交付过非常有用的东西吗？如果尝试使用一种不同的语言，你又能学到多少东西？有志向的学徒天生就是要跑到终点线，尽快地成为熟练工。但要记住，你正行走在“漫漫长路”上，这一旅程并不是一次短跑。花点时间好好利用自己的学徒期，要明白不管你已经编程三个月还是五年，说到软件工艺，你仍然只是个初学者。


Concrete Skills
Reflect As You Work
Stay In The Trenches
Breakable Toys
Unleash Your Enthusiasm
Create Feedback Loops
Be The Worst
Sweep The Floor
The Deep End
Nurture Your Passion
Find Mentors
Rubbing Elbows
Record What You Learn
The Long Road
Kindred Spirits

第5章

恒久学习

如果我们放纵自己，我们将总是需要等待一些消遣或其他事情结束才能安心工作。只有那些对知识非常渴求，以至于在不利的环境下仍能坚持探索的人才会取得更大的成就。从来就没什么“有利条件”。

——C.S. Lewis，“Learning in War-Time”（战时学习），“The Weight of Glory”（荣誉的分量）及其他演说

Andy Hunt，一位很受人尊敬的软件技师，曾多次表明一种观点：软件开发由两种主要的行为构成：学习和交流（《Pragmatic Thinking and Learning》[注重实效的思考和学习]，第3页）。基于这种思想和观点，我们可以得出：学徒期的主题就是学习，成功学徒的显著特点就是能证明自己的学习能力。学徒总是渴望有机会用技能替换自己的无知。当一个学徒面对他必须应付的工作复杂度，还有看似会把人淹没的信息量，这绝非易事。除了学习“具体技能”这种基础的活动，学徒还必须学会如何学习，显然，向着熟练工的转变过程并不会消除学习的需要。师傅们的特点之一就是：他们愿意放下辛苦得来的特定领域的专长，以便学习新东西。对那些为精通技能而长途跋涉的人们，学习是一种恒久的行动。

跟“恒久学习”有关的模式可应用在你的整个职业生涯中，但考虑到学习对于学徒的重要性，早一点把它们运用到职业旅途中非常重要。对想要成长更快的学徒来说，“提高带宽”是一种基本行动，而且该模式还能促动其他几种“恒久学习”模式，如“质脆玩具”、“使用源码”和“不断实践”。所有这三种模式都来自于对新信息的全面接触，或者对获取新知识的渴望：你是否在实践一种新技术，为学习新平台而构建一些东西，或者在研究一种革新性开源工具的源代码。这个相对具体的模式之后是一些更“软”的自我发现（self-discovery）模式，从“且行且思”开始，到“记录”和“分享”你所学到的东西。有两种关键的模式将伴随你一直到学徒期之后的岁月，它们是“建立馈路”和“学会失败”。为了转变成熟练工并最终成为师傅，你需要善于建立反馈回路（feedback loop），也需要对自己的弱点了然于胸。

提高带宽

学会那些本来不会做的事情，
常常比去做已经会做的事情更加重要。
——Jim Highsmith，《Agile Software Development Ecosystems》
（中译本《敏捷软件开发生态系统》，机械工业出版社）

情景分析

你已经掌握了一组基本技能。

问题描述

你对软件开发的理解较为狭隘，只关注日常工作中的低层次细节。

解决方法

你一直像使用吸管喝水那样慢慢地汲取知识。但在学徒时期，有时候你必须快速地吸收那些就像从消防水管喷涌而出的、大多数软件发者都能获取到的知识。

对学徒来说，提高获取新知识的能力是关键一步，尽管有时知识会多得让人崩溃。你必须开发一些必要的方法和技巧来高效地获取、理解、维持并应用新知识。该模式不只是要求你读一本书来了解软件开发中你所不熟悉的某个方面。它包含从多个维度去寻求新的知识和经验。比如：

- 注册成为博客聚合器的用户并开始订阅软件开发方面的博客（blog）。使用现代的机器翻译技术，你甚至没必要把自己限制在英文博客上。你可以遵循Tim O'Reilly[译注1]的建议，跨多种不同的技术领域去跟踪他称之为“阿尔法奇客”（alpha geek）的人。[注1]这些人不一定是最好的程序员，但总体上讲，他们往往能比我们提前多年感知到新的技术趋势。可以考虑撰写自己的博客来审视你挑选的博客所涉及的主题。
- 从Twitter上跟（follow）一些软件大师，留意他们正在做什么。
- 订阅流量较高的在线邮件列表，重现别人遇到的问题并尝试回答他们的提问。

注1：Tim O'Reilly，“Watching the 'Alpha Geeks'：OS X and the Next Big Thing”（关注“阿尔法”奇客：OS X和下一个大事件），参见：*http://www.linuxdevcenter.com/pub/a/mac/2002/05/14/oreilly_wwdc_keynote.html*。

译注1：Tim O'Reilly，O'Reilly Media的创始人，开源运动的支持者。

- 加入一个新成立的对某种新技术比较兴奋的本地用户组。参加时不要老是保持沉默——向小组介绍自己，并向别人提供帮助。
- 说服老板送你去参加技术大会。即使他们不愿意付钱让你参加，你也可以从网站上阅读幻灯片，并下载演讲的音频或视频。
- 在读完一本书之后，写封简短的邮件与作者联系一下，表达一下谢意，问一点问题。作者、演讲者和大师们看起来会让人生畏，或者看起来很忙，但其实他们常常很乐意跟读者通信交流。
- 不要忘了网上有成百上千的在线教程、播客（podcast）和视频（如Tech Talks的广泛系列），可以从iTunes和YouTube免费下载。

随着学徒期优先级的转移，你最终需要关闭这个消防水管，从而能专注于项目工作。但在学徒期中，至少应该有那么一段时间你在使用这一模式。这很重要，不仅因为你在这段时间所获得的知识，还因为它本身就是一种需要开发的技能。熟练工和师傅也会寻找可以运用该模式的机会，来推进他们的职业生涯和职业技能，特别当他们想进军新的技术领域时。

Dave和消防水管

2000年年末，当老板给我一次学习Perl语言的机会时，我便立即开始提高自己的带宽。我觉得自己必须补充很多知识才能跟上，于是在读完几本Perl书之后，便开始寻找所有能让我学到更多东西的机会。我决定让自己尽可能快地达到下一个水准：成为一名Perl开发者，而且我知道每次只读一本书无法让我足够快（我很强，不可以么？）。因此，我加入了*http://perlmonks.org*，在comp.lang.perl.misc上面提问并回答问题，参加了几次Perl Mongers[译注2]会议，并开始玩Perl Golf[译注3]（是的，我很强）。这样过了大约一年，我不得不放慢自己摄入知识的速度，以便使自己神志清楚，并准备结婚。但我的确取得了进步，而且有了更多可以支配的资源来应付遇到难题的时候。

译注2：Perl Mongers是国际Perl用户组织的一个较为松散的协会。

译注3：Perl Golf是一种竞赛，比赛内容是找出能解决给定问题的最短Perl代码。

之后，在2002年春天，我看了Kent Beck写的《Extreme Programming Explained》（解析极限编程－拥抱变化），并从中看到了一次让自己的成长超越Perl，进入测试驱动开发、极限编程、面向对象设计和设计模式世界的机会。再一次我提高了自己的带宽，读了一大堆优秀书籍，开始参加本地的一个敏捷软件开发用户组，自付路费去参加一次“极限编程/敏捷开发宇宙”（XP/Agile Universe）大会（很幸运，那一年的会议地点离我家很近），加入了极限编程邮件列表，开始阅读相关的博客，之后又开始自己撰写博客。这一段提高带宽过程的产出为我赢得了一份在ThoughtWorks的工作，它是一家跨国的敏捷开发咨询公司。ThoughtWorks带给我的学习机会永远地改变了我们职业生涯和学徒期。

快到2005年年底，那正是我从一名学徒开始转向熟练工的时候，我又看到了一次机会：Ruby on Rails正在登上软件开发的舞台。这次机会使我加入了Obtiva，一家更适合我生活方式的本地咨询公司，在那里我创立了Obtiva的软件工作室，并启动了Obtiva的学徒训练项目。

当你懂得了怎样为自己“提高带宽”，下一步就要理解何时“提高带宽”。在收集并使用新知识的过程中你可能会变得迷茫，特别在这个行业里，获取那些作品丰富的思想家们的最新思想变得越来越容易。一些人会迷失在有趣信息的海洋中，永远没有回到实际的软件制作上来。虽然“提高带宽”充满了乐趣，而且本身就是一项有趣的技能，但对学徒来说，它是到达终点的手段。要审慎地使用这一模式，因为它虽然能加速你的学习，也会降低你的开发速度，因此如果连续使用几个月以上，收益会越来越小。

行动指南

在接下来的几个月里参加一个本地用户组。研究一个你可能参加的相关国内会议。开始阅读一本由会议上的某个演讲者写的书。读完之后准备一些问题与作者联系。

不断实践

我们所知道的大师，不会仅为了做得更好而让
自己专注于某项技能。实际情况是：
他们热爱实践——并且正因为热爱实践所以才做得更好。
事情总是相辅相成的，做得越好，
他们就越喜欢反复不断实施这些基本的实践活动。
——George Leonard，《Mastery》（精通）

情景分析

你想让自己在某些事情上做得更好，你想在新的领域开发“具体技能”。

问题描述

你的日常编程活动不会给你通过犯错来学习的机会。似乎你一直都在台上演出。

解决方法

在一个可以很轻松犯错误的环境中，花点时间不受干扰地实践你的技艺。

在理想的世界中，我们可以采用K. Anders Ericsson的研究中所描述的“刻意实践”（deliberate practice）技术，指导者基于对你能力和弱点的了解，给你布置一个练习。等这个练习结束，指导者基于一种目标标准和你一起评价你的成绩，然后和你一起设计下一个练习。指导者会基于她指导其他学生的经验来设计新的更具挑战性的练习，这些练习将促使你反思自己的技能，找到更有效的工作习惯，并开发出基于更加抽象的知识“块”来“看”问题的能力。久而久之，这一连串的练习会磨炼你的力量，纠正你的弱点。悲惨的是：我们并不生活在一个理想的世界中，学徒们必须借助自己的资源来达到同样的效果。

> 在软件开发中，我们是在工作中实践，这也是我们会在工作中犯错误的原因。我们需要找到把实践与职业分离开来的方法。我们需要实践期。
>
> ——Dave Thomas[注2]

“注重实效”的Dave Thomas借用了武术中的概念，提出了编码路数。“路数”（Kata）是师傅为了让徒弟对武术基础心领神会而专门设计的一系列动作。“路数”是在没有对手的情况下练习的，强调的是流畅性、力量、速度和控制力。Dave Thomas在他的博客中发布了一些“路数”，鼓励技师们通过“路数”来实践。

Laurent Bossavit和一组巴黎的软件开发者将武术的比喻进一步发扬，他们建立了编码人员的“武馆”（dojo），一个人们可以定期聚会并公开练习编码“路数”的地方。受巴黎编码“武馆”的启示，Bob Martin“大叔”也在自己的博客中发表了一些“路数”，来宣扬工艺实践的好处。

> 初学者学习靠的是动手，而不是说教。他们实践，实践，不断地实践……通过不断反复这些同样的练习，我们增强了自己的技能，训练自己按照TDD和简单设计的原则对问题做出反应。我们一遍又一遍地重新排布自己的神经细胞，使其按正确的方式做出反应。
>
> ——Robert Martin[注3]

显然，编码“路数”只是“不断实践”的方式之一（“质脆玩具”是另一种）。这一模式的关键是划出一些时间，在一种没有压力、又充满乐趣的环境中开发软件：没有发布日期、没有产品问题、没有外界打断。Dave Thomas如此评说实践：“它必须让人感到轻松，因为如果你不能放松，你就不会从实践中学到东西。”

实践过程需要结合较短的反馈回路。虽然理论上实践是有益的，可如果

注2：Dave Thomas谈编码路数：*http://codekata.pragprog.com/2007/01/code_kata_backg.html*。

注3：Robert Martin谈编程“武馆”：*http://butunclebob.com/ArticleS.UncleBob.TheProgrammingDojo*。

你得不到周期性的反馈，你很可能养成坏习惯。这正是编码“武馆”最有意义的地方：在紧密联系的技师社区中公开练习。作为一名技师，随着你的不断成长，这种对于持续反馈的需要会慢慢降低，并逐渐被高级学徒所应承担的责任所替代：与经验更少的开发者一起实践，并在此过程中培养好的习惯。

George Leonard所描述的那些师傅们之所以喜爱实践，原因之一是每次实践的时候，他们都做一点不同的事情。问题的关键并不是刷新记忆，而是在哪怕最简单的技能活动之间找出细微差别。也许你的祖母曾跟你说：“实践产生完美。”她说得不对。实际上，实践产生永恒。因此，要悉心关注你所实践的事情，经常评估一下，确保自己实践的东西没有过时。每天选择正确的事情来实践是一种技能，这几乎跟反复实践这一行为本身同样重要。在实践期间，确保自己拥有趣味练习的方法是涉猎一些老书，如《Programming Pearls》（编程珠玑）、《More Programming Pearls》（续编程珠玑），还有《Etudes for Programmers》（程序员练习曲）。它们都是很久之前写就的，足以确保它们别无选择，只能关注于计算机科学的基础，而不是最新的时髦框架。这些书的作者们都知道，深入领会算法复杂度和数据结构基础没有坏处，而且时时有用。这些主题也提供了一个几乎用之不竭的趣味小问题源泉，使你的实践期保持有趣、新鲜，并充满教育意义。

行动指南

从前面提到的某一本书中找一个练习，或者自己设计一个来做。确保所找的问题比某个你能轻易解决的问题困难，但只是困难一点。第一次你应该需要努力才能解决它。接下来的四个星期中，每星期都将这个练习重做一遍并观察自己的解决方法是如何演化的。作为一名程序员，你应该考虑一下，这些演化道出了你的哪些实力和弱点？运用这些发现，尝试找出或设计一个新的、能对自己的能力产生可观影响的练习。如此反复。

参考模式

“质脆玩具”（第5章），“具体技能”（第2章）。

质脆玩具

任何事情，除非你热爱它，否则你不可能真正做好它，
如果你热爱黑客那样的工作，
你将不可避免地去做自己喜爱的项目。
——Paul Graham，《Hackers & Painters》（黑客和画家）

我们都可以从编写随意的“玩具”程序中受益，
编写玩具程序时，我们会设置一些人为的限制，
从而将自己的能力推到极限。
——Donald Knuth，《The Art of Computer Programming》
（中译本及双语本《计算机程序设计艺术》，机械工业出版社）

情景分析

来自失败的经验与来自成功的经验同样多（如果不是更多的话）。

问题描述

你工作在一个不允许失败的环境中。然而失败常常是学习一样东西的最好方法。只有通过尝试大胆的事情，失败，并从失败中学习，然后再尝试，我们才会成长为那种面对困难也能成功的人。

解决方法

设计并构建一套玩具系统，此系统从使用的工具集上（而不是功能范围上）与你在工作中构建的系统类似。通过这种方式为失败做出预算。

如果来自失败的经验可以与来自成功的经验一样多，那么你需要一个相对私有的空间来寻求失败。在抛球杂技中，抛三只球的表演者，如果从来没抛过五只球，那他就永远不会取得进步。而那些连续几个小时去拣

落下的球直到拣得背疼的人，最终却能把技艺练好。同样的经验也适用于软件领域。正如抛三只球的表演者不会在正式的表演中抛五只球，软件开发者也需要一个安全的地方来犯错误。

Steve Baker是一位在加拿大新斯科舍（Nova Scotia）省工作的青年，在那个不大的软件开发机构中，他担任主管和专家。Steve曾讲述过这样的期望如何影响了他："每个人都期待我已经知道了解决问题的正确方法。既然不能通过那些项目来获取经验，我不得不停止学习。"这与Ade的咨询经验类似，在咨询当中，他也犯不起错误，并且不能从那些依赖他永远正确的人的身边走开。Ade知道，为了学习他需要有允许球落下的自由。跟他所面对的无数软件开发者一样，Ade通过"质脆玩具"来帮助自己学习。

在实施"质脆玩具"模式时，要让玩具系统跟你的学徒生活相关而且有用。比如，构建自己的wiki、日程表或者地址簿。针对要解决的问题，你的方案可以是过度工程化了的，而且很可能有一种现成的东西可以轻易取代它。然而，这些项目是允许你犯错的。在这些项目中，你可以尝试那些可能导致灾难性失败的思想和技术。但这些失败只可能伤害到一个人，那就是你。

运用这一模式的经典例子就是那许许多多构建了个人wiki的人。对学徒来说，个人wiki是个了不起的工具，因为你可以用它来"记录所学"。Wiki可以成为很好的"质脆玩具"，因为它们可以很简单[注4]，而且你可以参考"使用源码"这一模式，去找出无数的例子来看。随着时间过去，维护一个wiki可以教会你关于HTTP、REST[译注4]、文本解析、Web设计、缓存、全文搜索、数据库和并行等各种技术。如果维护它的时间足够长，当你最终增加了一种特性，需要改用不同的存储格式却不想丢弃所有数据时，它还能教你有关数据迁移的技术。

"质脆玩具"模式的其他例子包括像Tetris和Tic-Tac-Toe这样的游戏

注4：　最短Wiki竞赛：*http://c2.com/cgi/wiki?ShortestWikiContest*。

译注4：　Representational State Transfer，针对万维网这样的分布式超媒体系统提出的一种软件架构风格。

（我们的一位前同事有一种习惯：每学习一门新语言，就用它来编一个游戏）、博客软件和IRC客户端。问题的本质在于构建玩具包含了对新事物的学习，而且提供机会让你在特殊的环境中加深对手中工具的理解，这个环境不仅安全（因为你是唯一或者最重要的用户），而且，即使跟最强大的商业产品比起来，你仍有余地来更好地服务自己作为一名用户的需求。

这些就是你的“质脆玩具”。当你带着它们从一份工作转向另一份工作，其中的某一些将成为自身技能的生动体现。尽管如此，你还是要记住：它们只是玩具，而且正因为这一点它们应该是有趣的。如果它们没什么意思，那么当最初的热乎劲过去之后，它们将成为尘封的旧物，而你则将自己的精力关注到你真正乐于构建的东西上去了。

这类玩具经常都是些工业级工具的简单再实现，重新实现它们可让你更深入地理解哪些因素导致了这种工具的现有设计。甚至会有这种可能：你的某个玩具获得了自己的生命，并拥有了其他用户。在那种情况下，你最后将不得不寻找一个新的质脆玩具。

Linus做了个玩具操作系统

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?（主题：在minix中你最想看到什么？）
Summary: small poll for my new operating system.（概要：为我的新操作系统做个小调查。）
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT （日期：1991年8月25日 GMT 20:57:08）
Organization: University of Helsinki（组织：赫尔辛基大学（赫尔辛基是芬兰首都，是Linux之父Linus的故乡，Linus当时是赫尔辛基大学的学生））

使用minix的朋友们，大家好！

我正在做一个针对386（486）AT机型的（免费）操作系统（只是业余爱好，不会像gnu那样很大很专业）。我从四月份开始就在酝酿这件事，现在差不多好了。我想得到有关人们对minix系统中各种好恶的任何反馈，因为我的系统有点像minix（相同的文件系统物理布局，缘于一些现实的原因；以及其他一些相同的东西）。

> 目前，我已经移植了bash（1.08）和gcc（1.40），它看起来可以工作。这表明我将在数月之内做出一些实用的东西，我想了解大多数人想要哪些特性。欢迎任何建议，但我不承诺一定实现。：-）
>
> Linus (torvalds@kruuna.helsinki.fi)
>
> 附笔：是的——所有的minix代码都是免费的，而且它还有支持多线程的文件系统。但它不可移植（使用了386的任务切换等机制），而且可能永远不会支持除AT硬盘之外的其他东西，因为我只有这些。：－（

“质脆玩具”模式与“只求最差”很像，但后者是关于如何寻找一个可让自己成长的团队。“质脆玩具”更多的是关于如何跨出自己的技能边界并独立构建完整的软件项目，以此来刻意创建学习机会。该模式与“白色腰带”和“正视无知”也有联系，但它较少关注于放下以前的知识。

行动指南

使用你最喜欢的工具构建世界上最简单的wiki，同时保证最高的质量。最初的版本能有一个简单的用户界面，让你浏览并编辑纯文本文件就可以了。随着更多的时间投入，你可以增加更多特性，并找出一些有趣的方法使你的wiki与已有的成千上万种wiki有所不同。不要被已有的实现所约束；相反，让你的专业兴趣指导你的实现。比如，你可能对搜索引擎有兴趣；这时，你的wiki中可以实现高级算法或者标签机制。你到底决定做什么完全不重要，只要你实验、学习。

参考模式

“只求最差”（第4章），“正视无知”（第2章），“记录所学”（第6章），“使用源码”（第6章）。

使用源码

> 为[成为一名程序员]做准备，最好的方法就是写程序，
> 并学习其他人写过的优秀程序。我当初所做的是：

到计算机科学中心的垃圾筒中，去寻找他们操作系统的清单。

——Bill Gates，《Programmers at Work》（工作中的程序员）

情景分析

初涉开源世界的新人常常发现别人用这样的话回答他们的问题："使用源码，Luke。"这表达出了与软件有关的一个基本事实：代码是最终的裁决者。如果代码与程序员的意图不一致，那后者就什么也不是。一个人只有通过阅读代码才能真正理解一套系统。

问题描述

如果没有好的实践范本拿来研究并效仿，"不断实践"模式就只会保护那些你都不知道自己已经养成的坏习惯。如果你从没穿着别人的鹿皮靴走上一英里，你会以为所有的鞋都会灌进沙石。如果你周围的人没有能力区分好代码和坏代码，你如何能认识到自己工作中哪些地方做得好呢？

解决方法

找别人的代码来读一读。从你日常使用的应用和工具开始。作为学徒，有一种想法会支持你阅读代码的信念：那些构建了你所用工具的人多少会有点与众不同，或有些特别，或比你更厉害。阅读他们的代码，你可以学会像他们那样写程序，而更重要的是，你将开始理解创造了你所用工具基础架构的那些思考过程。

在研究一个开源项目时，要养成下载最新版代码的习惯（最好从它们的源码控制系统下载），这样你可以查阅它的历史，跟踪未来的发展。看一下代码库的结构，想一想为什么代码是那样组织的。看一看开发者组织代码模块的方式是否有一定的道理，拿它跟自己可能使用的方式比较一下。

试着重构代码，从而理解为什么它的编码者做了那样的决定，同时想想如果你是那个编码的人，写出的代码将是什么样子。这不仅能让你更好

地理解这些项目；还确保你能构建这些项目。如果你发现了做一件事的更好方法，你就完全可以为这个项目贡献代码了。

在研究代码的过程中，你会不可避免地看到一些自己完全不同意的决策。问问自己，是否项目的开发者可能知道一些你不知道的东西，或者相反。考虑一下有没有可能这是一个历史遗留设计，需要重构一下；并考虑：为相关特性做个玩具实现是否有助于澄清问题。

除了阅读他人的代码（并在别人需要时提供反馈），也试着找找周围有没有人有兴趣阅读你写的源码。若能拥抱他们的反馈，并从中滤除纯个人的偏好，你将成为更好的程序员。还要记住：要成为熟练工，你必须帮助其他人成长，因此你也应非常开放地阅读他人的源码。

在我们访谈过的程序员中，有种常见的做法是加入一个实施“代码复查”（code review）或者“结对编程”（pair programming）实践的团队或项目中。这类实践创造了一种环境，使得程序员可以很自然地花时间阅读其他人的代码，让其他人阅读自己的代码，并相互学习。这样的团队更容易造就水平极高的程序员。其他一些环境，比如多数高校的计算机科学系，容易忽视实际工作中的程序员在阅读代码上花的时间比撰写代码更多这一事实。他们之所以这样，是因为让每个学生重新发明轮子能造出一些更容易评分的作业。然而，训练自己使自己更善于那些占用更多日常工作时间的任务（阅读代码）是对个人效益的优化，从长期来看能获得更高的回报。即使管理这种工作环境的非程序员们不欣赏这类实践，也改变不了这一事实。

通过阅读其他人编写的各种好的、坏的以及无关紧要的代码，你会慢慢学会特定语言社区中的各种惯用法和精妙细节。久而久之，这会开发你从别人编写的代码中推测其意图的能力，还会让你学会如何应付代码和意图不一致的情况。这些技能会使你成为团队中更有价值的部分，因为你将能够直接维护别人的代码，而不必因为看不出代码的用途而不得不每次重写它。

最终你将获得一个工具盒，里面都是从其他人的代码中收集来的各种奇

技。这会磨炼你更迅速、更快捷地解决小问题的能力。你将能处理别人认为不可能解决的问题，因为他们接触不到你的工具盒。

看一看Linus Torvalds编写的Git分布式源码控制系统的代码，或者任何Daniel J. Bernstein（众所周知的djb）编写的代码。像Linus和djb这样的程序员偶尔会用到我们大多数人甚至从未听说过的数据结构和算法。他们并不是魔术师——跟大多数人相比，他们不过是花时间构造了更大更好的工具盒。开源的优势就在于你可以随意观看他们的工具盒，而且可以把他们的工具变成你自己的。

软件开发领域的问题之一是缺少教师。不过多亏了诸如sourceforge.net和GitHub这些站点上开源项目的繁荣，你可以从全世界程序员社区中那些相对有代表性的代码范例中学习。跟传统的教学不同，它们并不是为演示一个技术点而设计的玩具项目，其中不会有各类捷径，以及当问题变得困难时的所谓“留给读者的练习”。它们都是解决实际问题，而且不断演变的真正项目。你可以在项目的开发者不断学习并适应新需要的过程中跟踪项目的进展。通过研究实际代码的演进方式，你可以不必亲手构建那成百上千的软件项目也能更好地理解某些设计决策的作用。这提供了从别人的错误中学习知识的机会，以及获得比简单的代码阅读更关键的技能——无师自通能力——的机会。

在《Programmers at Work》一书中，Bill Gates说：“对编程能力最精细的测试就是给程序员大约30页的代码，看他能多快地通读并理解它。”他意识到了一种很重要的东西。那些能直接从源码中快速汲取知识的人很快就能成为更好的程序员，因为他们的老师就是世界上的每一位程序员写下的一行行代码。

> 要学习模式、惯用法和最佳实践，最好的方法就是阅读开源代码。看看其他人是如何写代码的。这是一种保持自己不落伍的优秀方法，而且是免费的。
>
> ——Chris Wanstrath在2008 Ruby大会上的主讲内容[注5]

注5：2008 Ruby大会主讲。视频：*http://rubyhoedown2008.confreaks.com/08-chris-wanstrath-keynote.html*。手稿：*http://gist.github.com/6443*。

行动指南

挑选一个算法精深的开源项目，如Subversion、Git或Mercurial这样的源码控制系统。浏览项目的源码，记下让你觉得新奇的算法、数据结构和设计理念。然后写一篇博客，描述一下项目的架构，着重突出自己学到的新思想。你能在日常工作中找到可运用同样思想的场合吗？

参考模式

“不断实践”（第5章）。

且行且思

自我反省是困难的，但我相信我们能从失败中
学到的东西比从成功中学到的更多。
——Norm Kerth，《Project Retrospective》
（项目回顾：项目组评议手册）

情景分析

任何人，只要具有相当的能力，都会在多年之后发现自己被推上提升的阶梯。迟早你会在一个公司团队或者开源项目中戴上“资深开发者”的高帽。如果你不采取行动为这种提升做好准备，你可能突然发现自己成了Peter原理[译注5]的牺牲品：你被提升到自己“不胜任的级别”（level of incompetence）。

问题描述

随着你装进肚子里的工作年限和项目经历越来越多，你发现自己在等待一种质变，使你神奇地变成“经验丰富”的开发者。

译注5：The Peter Principle，“在等级式的组织结构中，员工往往被提升到他不胜任的级别。”，Laurence Peter博士等人1969年提出。

解决方法

在软件开发领域做一名会思考的从业者。这包括经常反思自己的工作状况。考虑一下自己的实践是新颖的，创新的，还是过时的。对那些团队中其他人都想当然的事情，多给自己画几个问号。如果觉得目前工作中有一些让人特别痛苦或开心的事情，问问自己事情是怎么变成这个样子的，如果问题是负面的，如何能改善它？我们的目标是通过将每一种经验拆分开，然后再以新的、有趣的方式组合起来，从其中提取最多数量的教育价值。

一种可用于明确表达这种反思的技巧是使用“个人实践图”（Personal Practices Map）。这是Joe Walnes在伦敦的“极限编程周二俱乐部”上介绍的一种想法。它要求人们有意识地写下自己所做的事情以及这些事情之间的联系。在每个人写下他们的实践之后，整个小组对其中的实践展开讨论。如果你看看那个“人们的个人实践图”的网页，[注6]你会看到由Ade和其他几名开发者画出的图。

反复使用这一技巧，会得到这样的结果：你的一组实践中所产生的变化被凸现出来。比如，2003年以后，Ade从“从不使用调试器”转变到实践“测试驱动的调试”，后来在实现复杂算法时又开始刻意使用不变量（invariant）。拥有一张看得见摸得着的实践图表，能使你更深入地思考自己所使用技术的每一次变化。在Ade的例子中，采用测试驱动开发使他重新评估了其他所有的实践，而图表成了将这一变化形象化的工具。

这种观察、反思并改变的过程并不只限于你自己的行为。你还可以悄悄地观察团队中的熟练工和师傅。思考他们使用的实践、过程和技术，看看这些东西能否跟自己经验中的其他部分关联起来。只需要在更有经验的技师们着手工作时近距离观察他们，即使是学徒，也能发掘出新颖的思想。

注6：*http://www.xpdeveloper.net/xpdwiki/Wiki.jsp?page=MapsOfPeoplesPersonalPractices.*

2004年，Dave是一个极限编程（XP）团队的一员，那个团队里有好几名世界级的开发者。他们拥有一种相当标准的结对编程风格：一名开发者写一个测试，然后把键盘推给他的搭档，他的搭档写代码让这个测试通过，然后立即写出下一个测试，再把键盘推回给第一个家伙。第一个人写代码通过这个测试，如此反复。这种结对编程风格从未被真正讨论过；它只是在个别的经验中形成的。

Dave加入了他的下一个项目，在他向新队友们解释这种结对编程的风格时，他意识到这种风格需要有一个名字。Dave为此写了一篇博客，结果引起了一连串的反应，很快StickyMinds.com邀请他写几篇专栏文章。所发生的这一切仅仅因为Dave反思了更高级的开发者引入的实践。

敏捷社区采纳了这一过程的某个版本。由Norm Kerth的《Project Retrospectives》一书推动，它要求团队周期性地聚在一起回顾项目的状态，以找出改进的方法。这样一来，它就比学徒可能采取的持续自我分析更加正式。它也要求相对开明的管理者愿意提供一个相对安全的环境，理由是尊崇Kerth的最高指导原则："无论我们发现了什么，考虑到当时的已知情况、个人的技术水平和能力、可用的资源，以及手上的状况，我们理解并坚信：每个人对自己的工作都已全力以赴。"[注7]

学徒们并非都能有幸在这样的环境中工作，但即使在相对不那么宽松的企业文化中，养成富有成效的反思习惯也是有用的。

只要你工作的时间足够长，人们就会开始称你为"老手"，但这不应该是你的目标。所有的经验都表明你已经有能力在这个行业生存。但它显示不出你所学到的知识量，仅仅是你所花费的时间。在我们行业的某些领域，有时很容易将同样的时间经历重复10次却没有取得能力上的实质进步。事实上，这有时会变成"反经验"（anti-experience），即这样一种现象：每一次新的时间经历仅仅强化了你所养成的坏习惯。[注8]这就

注7：The Retrospective Prime Directive（回顾活动最高指导原则）：*http://www.retrospectives.com/pages/retroPrimeDirective.html*。

注8：反经验（Anti-Experience）：*http://c2.com/cgi/wiki/changes.cgi?AntiExperience*。

是为什么你的目标应该是达到技能娴熟，而非“有经验”。技能水平的提高，是你在研究、适应及改善工作习惯方面所付出努力的唯一有效的证明。

行动指南

为你的工作习惯画一张“个人实践图”。重点关注一段时间内没有改变的那些实践之间的关联。问自己，如果你发现当中的某种实践实际上对生产率起反作用，你的图该做怎样的修改。仔细检查其中的某一种实践，看看是否存在达到同样目标的其他方法。它未必是更好的方法；只要不同就够了。然后问自己，如果你采用其中一种不同的实践，这张图又会怎样变化。

记录所学

你也不应该低估写作本身的威力……
你会失掉更大的目标感。但写作能让你后退一下
并完全想通一个问题。即使最怒气冲冲的演说
也能使一位作者达到某种程度的深思熟虑。
——Atul Gawande，《Better》（更好）

情景分析

你一遍又一遍地学到同样的经验。似乎没有一样能持续下来。你常常发现自己重复地做着诸如搭建CruiseControl[译注6]，SQL层次建模，或者向团队介绍模式这类事情。你记得以前做过非常类似的事情，但具体的细节却想不起来了。

问题描述

不从历史中学习的人注定重复历史。

译注6：CruiseControl：一种持续集成工具以及可扩展地定制持续构建过程的框架。

解决方法

在日志、个人wiki或博客中为自己的行程做个记录。将自己学到的经验按时间顺序记录，这会给你所指导的那些人提供一点启发，因为它使你的经历更加明朗，另外也为你自己提供了可以利用的重要资源。使用此模式的人迟早会经历这样的时刻：搜索一个棘手问题的答案，结果搜索引擎给出了一个指向他自己wiki或博客的链接。

使用博客来记录学到的经验还有一种附带好处：帮你结识“同道中人”；而一个带有随机链接的wiki能让你看到自身经验之间的联系。

尽量避免落入写下经验之后就忘记它们的陷阱。你的笔记、博客或wiki应该是一个托儿所，而不是一座坟墓——经验应该从这次记录中降生，而不是到那里去灭亡。定期读一读以前写的东西，你就能保证这一点。试着在每次重读这些资料时都找到新的关联。这种创造性的复读过程能让你基于新的数据重新评估旧的决策，或者坚定正在摇摆的信念。这两种结果都不错，只要你别停滞就行。重读你的笔记，你可以不断变换自己的过去和现在，从而造就你的未来。

Ade使用了同一种wiki的两个实例，一个用于记录私人的想法，另一个记录那些想要与别人分享的想法。在拥有公开记录的同时再保留一套私人记录，这意味着你可以同时利用两个世界的优势。公开记录成为你分享自己所学经验并从更广大的社区中获得反馈的途径；而私人记录使你得以痛苦地面对自己取得的实际进步。同时拥有内在的反馈回路和外在的反馈回路能使你增强自信，确信你对自己有着正确的评价。

在学徒期中，Dave为“坚持阅读”的过程维护了一个文本文件，其中摘抄了所有对他的学习有重要影响的文字。几年下来，这个文件增大到包含500多条摘录，最终Dave决定把它上载到网上在线分享。[注9]后来，当Dave开始撰写文章以及本书的时候，这些摘录成了极好的引用源。

别忘了：你选择的维护记录的工具也可以是一个重要的“质脆玩具”。

注9：*http://redsquirrel.com/dave/quotes.html.*

这一模式与“分享所学”类似，但“分享所学”的重点是：提高诚实、谦逊的交流能力，为成为熟练工铺路。而这里的重点则是记录下你通往技艺精通的路线，以便以后可以从中获取新的经验。

行动指南

拿出一本笔记簿，开始简单记录你对于本书的想法，或者它所激发的任何思想。所做的笔记一定要有个日期。读完这本书以后，针对所学的其他东西，继续按同样的方式使用这本本子。经过一段时间，记下的条目就会成为博客、杂志文章，甚至一本书的基础。

参考模式

“质脆玩具”（第5章），“同道中人”（第4章），“坚持阅读”（第6章），“分享所学”（第5章）。

分享所学

我无法估量一个慷慨的灵魂能带来多大的好运。
看看你周围最幸运的人，那些让人羡慕的人，
那些看起来总是被好运砸到的人。他们做了什么使自己如此出众？
如果好运总是重复发生，那就不是简单的“好运”了。
就我所知的幸运儿们，他们总是有所准备，
总是忙于打磨自己的技艺，他们眼光敏锐，
让朋友进入自己的工作中，而且常常让其他人觉得
能处在他的周围是一件很幸运的事。
——Twyla Tharp，《The Creative Habit》（创造性的习惯）

情景分析

你已经做了一段时间学徒。你懂得了一些东西，人们开始把你当成一个知识源来看待。

问题描述

直到现在，你一直心无旁骛地关注自己作为一名技师的个人进步。要成为熟练工，你需要具备有效沟通的能力，以及培养别人，使之加速前进的能力。

解决方法

要在学徒期的早期就养成定期分享所学经验的习惯。形式可以是撰写博客，或者跟你的“同道中人”一起开展“便当会议”。也可以在技术会议上做演讲，或者为你正在学习的各类技术技巧编写教程。

一开始这会有难度。毕竟，你还不是一位师傅，甚至连熟练工都不是：然后你就应该等在那里，而让更有经验的人继续把他们自己推向前进吗？不是的，你会发现你的学徒同伴们更希望你们自己当中的一员来理清一些复杂问题。你可能只知道一点点范畴论（category theory）或者基于原型的编程语言方面的知识。因为你只知道一点点，你的解释将会简单明了直击主题，而且不会假设别人已经了解任何预备知识。这会使你的解释成为更好的解释。你会发现，当第一次学习某个特定主题或某项特定技术时，编写一套你本来希望别人写给你的教程是很有帮助的。

人们看重各人自身的学习，也看重大家谦虚地分享新知识——处在这样的社区中，是学徒期非常重要的一方面。它使得原本深奥难懂的知识领域一下子变得易于掌握，并为学徒们提供了说同种语言的向导。

此外，教别人是一种非常强大的学习方法——相对于学的人，这一点对于教的人来说或许更明显。俗话说：“一个人教的时候，两个人在学。”

该模式与“记录所学”的关联最为明显。记录下自己学到的东西，就更便于与别人分享它们。另一方面，该模式也带有一点风险：人们对你分享的东西未必领情。

有些东西是不能分享的，一定要记住知识还有个道德维度。在分享一种经验之前，要考虑这种经验是否完全属于自己。它可能是机密，或可能给他人造成损失。基于当前的情景你觉得显而易见的事情，或许正是老

板的“秘制沙司”，而作为学徒，你太容易忽视分享这种知识的后果（法律上的、经济上的或政治上的）。

还有其他一些经验，一旦分享就会损害你跟团队中的其他成员或者跟雇主之间的关系。如果其他人（不管是正确地还是错误地）认为你分享知识的方式不够谦虚，或者你分享经验是出于某种别有用心的目的，那么应用“打扫地面”模式所取得的收获会直接付诸东流。

“只求最差”能把你引向更好的学习机会，风险是你可能因此忽略了自己对工艺技术的责任。你不断寻找加速学习的机会，并因此落入一种无休止的自私地汲取知识的状态中，而从不考虑只要你“分享所学”就能从中受益的那些人。

Dave Smith的“铺平道路”（Prepare the Way）模式跟“分享所学”有紧密的联系。

> “铺平道路”为我们亮起绿灯，教育我们：作为开拓者，我们有更多的责任，在茫茫荒野开拓前进的过程中，我们有时要离开那条一直跟随我们的显眼的、安全的轨迹。
>
> ——*http://c2.com/cgi/wiki?PrepareTheWay*

行动指南

回想一下你上次学到的重要一课，整理成一篇博客，写出那些你当时希望拥有而且会帮助你学习的信息。

写完博客之后，想象你被要求为一场技术会议准备一次研讨，向其他人传授同样经验。为这次研讨列个提纲。看看当你在考虑“如何以一种引人入胜的方式教别人”时，你会不会重新思考自己学到的东西以及那篇博客。

参考模式

“只求最差”（第4章），“同道中人”（第4章），“记录所学”（第5章），“打扫地面”（第4章）。

建立馈路

处在软件行业的我们，工作中面对的是一种相对不可见的产品，
而正是这种不见性增加了我们对反馈的需求。
——Jerry Weinberg在Norm Kerth的项目回顾中

情景分析

你无法判断自己是否正遭受“意识不到的无能”之苦，因为恰如Justin Kruger和David Dunning在他们的同名论文中指出的：技艺不精的人常常不知道自己技艺不精。再者，越是技艺不精，你越不善于评估自己和他人的技能。成功和失败往往都会突如其来，而你所收到的那一点反馈只会成为对你自我评估能力的无情打击，而不会是帮你进步的支持机制。

问题描述

自我评估只能相对于你过去拥有的能力，而且永远缺乏客观性。你的团队很容易歪曲你对自身能力的判断。处在一个高于平均水平的团队中，这要么当你只是个候补歌手时就让你觉得自己像超级明星，要么当每个人看起来都比你更强的时候毁坏你的自信。另一方面，处在一个低于平均水平的团队中则会使你自鸣得意。即便使用“且行且思”模式，它也只能帮你分析过去，不会告诉你现在的情况。

解决方法

建立一些机制，定期收集关于自身绩效的相对客观的外部数据。通过尽早、经常而且高效地寻求反馈，至少你可以提高知道自己不行的概率。

有多种机制可用于获取反馈。在某种层面上，这包括使用像测试驱动开发和带有交互式解释器的动态类型语言这类技术，它们能使你的程序快速失败（fail fast）。另一种层面上，你可以通过代码复审和结对编程来获得反馈。考试和认证也可以成为有用的工具，来测定自己跟别人的

相对水平，虽然大多数情况下它们测出的只是你的应试技巧而非实际技能。另外一种获取反馈的方法是向别人询问他们对你工作的看法；比如，接触一下为新工作或职位提升而面试你的人，问问他们对你的看法。即使你没得到那份工作，只要别人准确地讲出他们为什么拒绝你，你也能从中得到很多。有时，这类反馈会揭示你个性中的一些自己原本不知道的（积极或消极的）方面。

如果你没有培养出对原始数据进行处理的能力，那以上所描述的所有机制都不会有任何用处。比如说，如果你的老板做年度审查，你必须能够去芜存菁才能得到有用的反馈。批评本身不太会成为有用的反馈，因为它没有告诉你别人期待的究竟是什么。其他类型的无用反馈诸如：更多关于别人而不是关于你的反馈（比如，“做这个吧，我在你这个年纪时也在做这个。”），完全伪装的建议，或者Dave Winer称之为“停止能量”（stop energy）[注10]的反馈。这通常表现为一些善意的建议，告诉你为什么你无法达到目标，建议你立即放弃而别再冒失败的风险。

那有用的反馈应该是怎样的呢？有用的反馈是这样一类数据：你可以基于它采取行动，而且它能针对某种特定行为给出多做或少做的选择。如果基于一种反馈你无法采取任何行动，那它就不是有用的反馈。或者至少它目前没用。如果环境变了，它可能突然变得有意义。记住以下建议：“如果你认为一样东西有趣，你就能学到有趣的东西。”（《Better》[更好]，第225页）

了解系统思考者（systems thinker）所指出的加强型反馈（reinforcing feedback）和平衡型反馈（balancing feedback）之间的区别也很重要。加强型反馈鼓励你多做某事，平衡型反馈鼓励你少做某事。结合这两种反馈，一套系统可以通过大量的小规模调整维持在一个相对稳定的状态。成功的学徒都能学会创造条件让自己快速地、经常地收到关于某种活动需要多做或少做的数据反馈。通常，这意味着学会交流思想，学会聆听而不打断别人。

注10：“What is Stop Energy?”（停止能量是什么），参见：*http://radio.weblogs.com/0107584/stories/2002/05/05/stopEnergyByDaveWiner.html*。

你需要练成不去维护自己当前知识水平而密切关注所有反馈的功力，在这方面该模式与“白色腰带”有所重叠。两种模式都强调了一种思想：学徒应该努力让自己“更可教”，从而潜在的“教师池”可以越来越大。

Patrick研究发现了不去获取反馈的后果

当我最终开始为主项目编写代码的时候，我实际上没有对Oracle框架以及别人设计的各种模块的工作方式有很好的理解。我只好把另一团队中一位资深工程师编写的代码用作我编码的模式基础，因为我们组的资深工程师们只给我很少的反馈。一段时间后，我发现自己实现功能的方法并不是开发那种应用的首选方法，于是我去找那位写之前代码的工程师，问他事情为什么会这样。我很幸运，那时我们还处在开发过程的“研究”阶段，所以我写的代码可以修正。

我遇到的最困难的事情是：当你正在犯错误时，愿意告诉你的人并不多；因此，成功的一半就是努力找到一个能尽快告诉你的人。回头想想，我觉得一个学徒不应该过早地致力于不犯错误，而应该尽早找出如何确定错误的办法。一旦学徒能确定他们的错误，从错误中学习就容易得多了。

——Patrick Kua，电子邮件

行动指南

从你的工作环境中找到一件你可以度量而且（更重要的）能够影响的事情。在一段时间内跟踪度量的结果。当它改变时，问自己这说明了关于你的什么信息。看看是否能用它（以及其他度量手段）来理解你对工作环境的改变所带来的效果。

参考模式

“且行且思”（第5章），“白色腰带”（第2章）。

学会失败

天赋常常被误解。它并不是超常的智力，
而是一种性格。它最需要的是一种承认失败、
不遮掩缺点并努力改变的意愿。它来自于刻意的，
甚至强迫性的对失败的反思，以及对新方案的持续探索。
——Atul Gawande，《Better》（更好）

情景分析

失败是不可避免的。它迟早会发生在每个人身上。实际上，在任何事情上都没失败过的人要么是不肯推进能力的边界，要么是学会了忽略自己的错误。

问题描述

你的学习技能增强了你的成功，但你的失败和弱点仍在。

解决方法

设法确定你常常会在哪些情况下失败并试着解决那些需要改正的方面。

这并不是要你沉溺于对过往失误的自悯，也不是一次追求完美的练习。真正的目标是让你对导致失败的模式、条件、习惯和行为有所自知。有了这种自知，你可以做有意识的选择，而且基于对自身能力边界和局限性的了解，在采用“自定路线”模式（第3章）时能使之趋于理想状态。

知道了那些使你失败的事情，你就可以在修正这些问题和减少损失之间做选择。要承认有些东西是你不擅长的，或者需要不成比例的时间和精力投入才能取得很少的进展。

这不仅能提供输入，使你做出更准确的自我评估，还能给你的目标加上现实限制。你不可能各方面都擅长，而承认这些局限是很重要的，因为

它迫使你有意识地应付注意力的分散并专注于自己的目标。这可能意味着你要承认自己永远不能抽出时间去攻读一个在职博士学位，或意味着你要放下旧的专家领域，因为自己已无法投入时间来维持那些技能。

比如，Ade在自己的私人wiki上设置了一组页面，其中罗列了他当前的技能集合，以及他的局限和能力边界。这可以帮他决定哪条边界应该向外推进（比如，尝试使用动态类型检查的语言来维护大型数据库）以及哪些地方应该停止浪费精力（比如，承认针对Commodore 64的6502汇编器不会再有大范围的使用了）。

行动指南

通过你选择的编程语言，使用一个简单的文本编辑器（后面你会看到为何在这个练习中不使用IDE很重要），一口气实现一个二分查找算法。暂时不要编译运行它。现在，写出所有你能想到的验证其正确性所需要的测试。记录下你在这一过程中发现的bug和问题。然后，同样不要编译运行测试，回到实现代码中去修正到目前为止你所发现的所有问题。重复这一迭代过程，直到你代码和测试都很完美并让你很满意。最后，试着编译并运行测试。大多数人这时都会发现一些之前没有想到的边角情况和一些琐碎的小错。在修正这些错误之前，尽量搞清楚为什么它们会发生，而且发生在一件你确信已经完美的事情中。这表明自己身上的什么问题？从你认为代码完美，到代码能实际编译并通过所有测试，其间的迭代过程中你所学到的所有东西都写下来。如果你觉得自己很勇敢，不妨再找个朋友来复查一下代码，看看她还能发现点什么。

参考模式

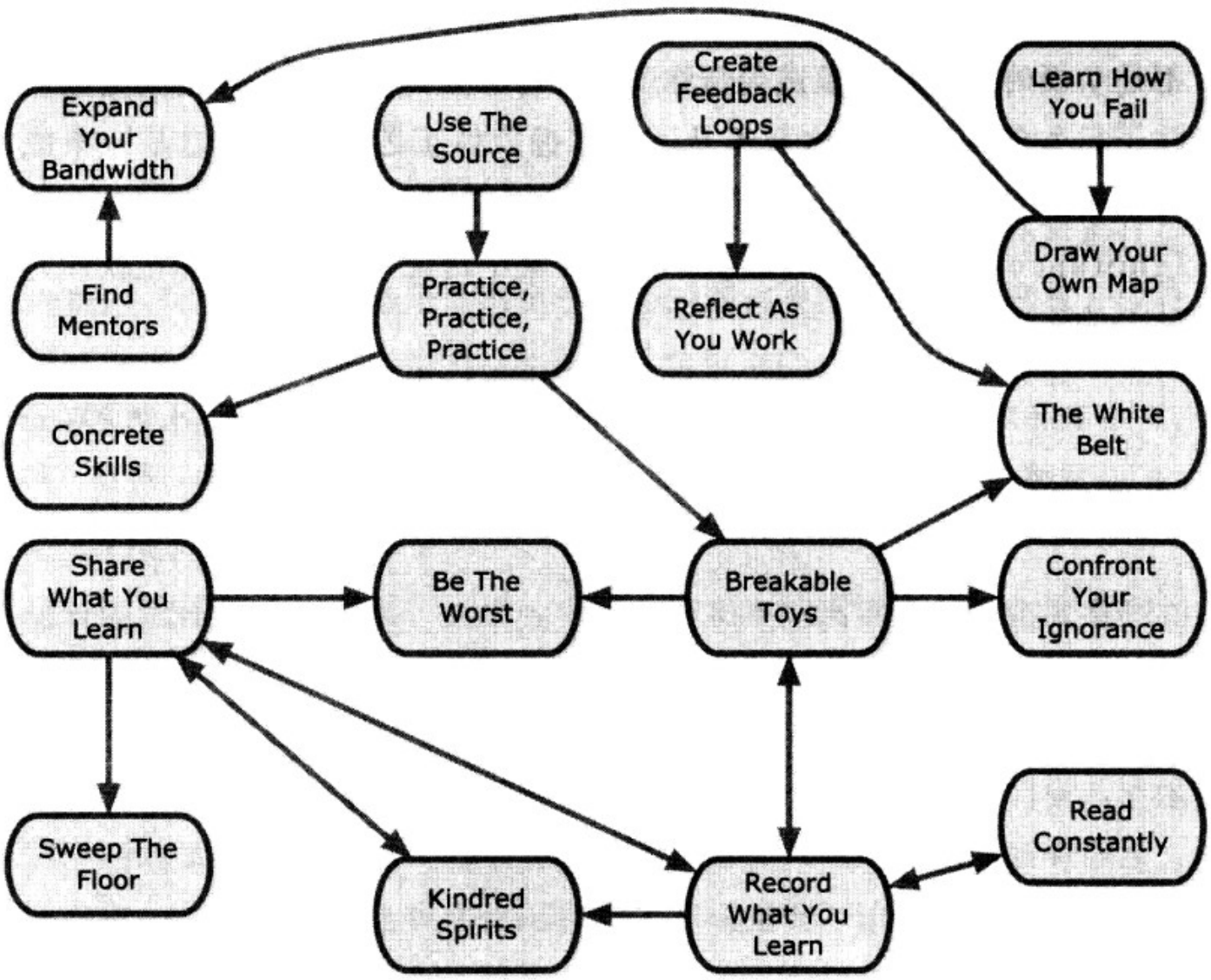
“自定路线”（第3章）。

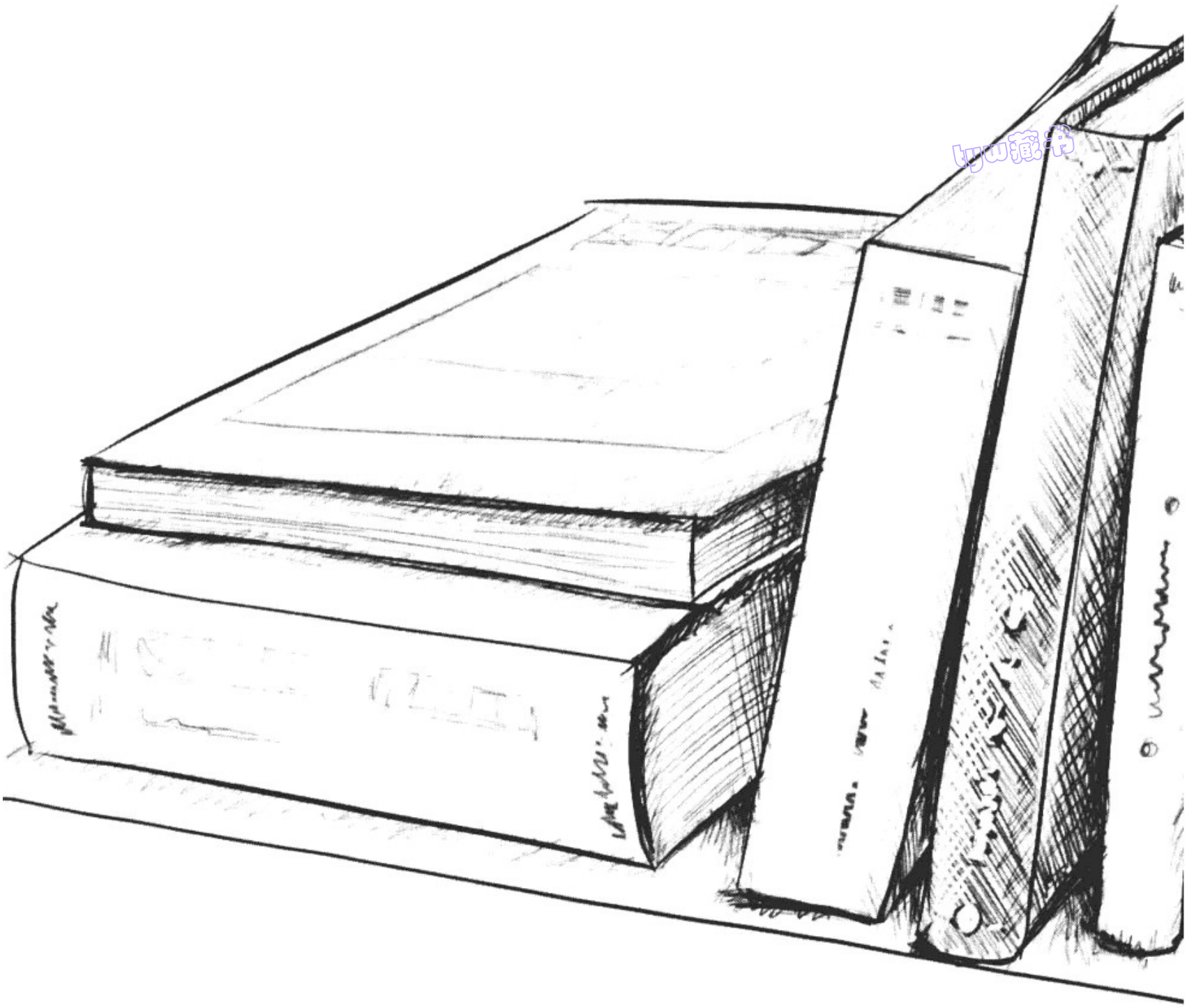
总结

你可以把“恒久学习”看作祝福，也可以看作诅咒。学习新东西是痛苦的，特别当你在顶着压力而且几乎没有人指导的情况下学习。尽管如此，正如运动员必须忍受艰苦训练后的肌肉痛楚，软件开发者需要忍受

学习新东西后的神智错乱。这种错乱有可能是进步的欢迎信号。自我反思，通过反馈回路找到不足，了解自己的弱点，所有这些从表面来看都是负面的，但这些模式会帮你削减自己的无知。另外一种做法是只专注于已知的东西，但这并不是通向掌握软件工艺的道路；它是一条成为单领域专家的道路。Phillip Armour在他的论文“The Five Orders of Ignorance”（无知的五重境界）中，集中讨论了无知和有知在软件开发中的作用。

> 软件不是产品，而是储存知识的媒体。因此，软件开发不是生产产品的活动，而是获取知识的活动。而有知和无知只是硬币的两面，所以，软件开发是一种削减无知的活动。


Expand Your Bandwidth
Use The Source
Create Feedback Loops
Learn How You Fail
Find Mentors
Practice, Practice, Practice
Reflect As You Work
Draw Your Own Map
Concrete Skills
The White Belt
Share What You Learn
Be The Worst
Breakable Toys
Confront Your Ignorance
Sweep The Floor
Kindred Spirits
Record What You Learn
Read Constantly



第6章

安排你的课程

他将不再是个靠学习成绩激励的人，而成为一个靠知识激励的人。
他不再靠外部的推动来学习。他的动力来自于内部……这种激励，
一旦抓住它，就会成为一种强势的力量。

——Robert Pirsig，《Zen and the Art of Motorcycle Maintenance》
（万里任禅游）

我们生活在一个信息充足的时代。印刷机的发明开创了一个新纪元：即使最贫穷的社会成员也能获取知识，从而获取力量，继而改变其生存环境。不断扩张的万维网和不断涌现的技术革新仍在持续降低我们想要获得的任何一种知识的门槛。因特网带宽不断提高，手持设备的数据储量似乎永远没有极限。如今，我们在任何时间、任何地点都能以文本、音频和视频格式访问高清晰度的媒体。和任何一个好学徒一样，你会喜欢使用最新最好的设备和媒体平台，但仍有一些信息，通常只能从普通旧式的书籍中获得。虽然博客可以提供阅读材料的极好源泉，有一些富有经验的从业者，如Jerry Weinberg、Fred Brooks、Steve McConnell和Kent Beck——从他们的书中所能获得的大量智慧却是无可取代的，即使再高带宽的信息也取代不了。即便你不是一只书虫，成功的学徒期仍然要有一些书，以及专用于学习的时间。可是，你已经离开学校了。不再有老师布置的阅读——你要靠自己去寻求建议，安排自己的课程。

阅读列表

没有人能一次学完所有的东西，但也没有哪种原理或规则
说学徒不可以今天学点这个，明天学点那个，
按照从来没人想过的次序学习新东西，或者学到某个点停下来，
然后转向其他的东西。学习某种特定过程，他不需要等到时间表
设好的时间；也不需要学习一种自己尚未准备好，认为没意思、
太吓人或不必要的东西。学习的人自己安排自己的课程。

——Howard S. Becker，“A School Is a Lousy Place to Learn Anything In”
（学校是个糟糕的地方，不可能学到所有东西）

情景分析

学到了足够多的能力与技艺，精通掌握了“入门语言”，在这之后，你开始四周观看，结果看到了自己仍需学习的不可思议的信息量。

问题描述

需要阅读的书籍数量快速增加，你不可能读完它们。

解决方法

维护一张“阅读列表”，用它来跟踪你打算阅读的书，并记下已经读过的书。

按照“分享所学”模式的精神，可以考虑把这张列表保存在公共空间。这能使其他人从你学到的东西中受益。我们使用的是http://bookshelved.org（Laurent Bossavit于2002年创建）上的wiki，但任何公共列表都可以。理想情况是，你的列表支持排序，并能区分哪本书是你何时读过的。

这一模式不只讨论如何管理自己想读的书，也是一种反映你以往阅读习性的机制。基于多年的数据，你可以在自己选择的学习资料中逐渐发现一些模式、趋势和缺口。这可以帮你更好地决定下一步读什么。如果把这种信息公开，其他人也会为你以后的阅读提供建议。这会使你发现隐蔽的联系和表面暗淡的珍宝。

任何一本书，你能从中获得的最有价值的东西就是一列值得阅读的其他书目。时间长了，你会发现某些书不断地从“参考书目”（bibliography）中跳出来，你应把那些书移到阅读列表的顶部。其他书会下沉。由于阅读列表实际上是个优先级队列（priority queue），最终你会发现有的书已经在队列中下沉得太深，你可能永远也不会再读它们了。这很好。这一模式的目的就是向你提供一种方法，为潜在的知识洪水排定优先级并进行过滤。

实施这一模式的主要难点是：你需要先对某个主题有深入的了解才能搞清楚哪些书需要读，按什么次序读。解决这一矛盾的方法之一是：开始时挑一本宽泛的书，让自己对目标主题有大致的了解；然后选择一些内容具体的书，从而掌握该主题中自己感兴趣的方面。另一种方法是依靠“同道中人”和指导者。指导者可以向你推荐一些必读书籍，与学徒伙伴们讨论则可以帮你弄清阅读它们的顺序。你还可以利用正在实施这一模式的其他人所提供的公开阅读列表。

另一难点在于搞清楚从哪里开始。你可以从微软出版社（Microsoft

Press）出版的《Code Complete》（代码大全）一书的第35章，或者《The Pragmatic Programmer》（程序员修炼之道——从小工到专家）一书的“参考书目”中获得相当不错的书籍列表，借此来构建自己的“阅读列表”。你还可以查看本书的“参考书目”，看看那些曾经启发过我们的书籍。

这一模式的构思要感激Ravi Mohan关于“书链”（Book Chain）[注1]的思想，以及来自Joshua Kerievsky的学习小组模式语言中的“顺序学习”（Sequential Study）模式。[注2] “书链”讲的是如何请别人向你推荐一套能将你带入新主题的图书，这一模式更多的是讲如何管理自己感兴趣的一系列书籍。该模式与“顺序学习”也不一样，因为它并不要求按照这些书籍相互影响的年代次序来阅读它们。在这一模式中，能使你在旅程中走得更远的那本书，就是你下一步该读的。

记住这是你自己的阅读列表，这一点非常重要。让其他人的建议影响你，这很好，但只有你自己真正清楚你目前的状况。因此，你才应该是那个对自己下一步学什么做出决策的人。除此之外，在合适的时间阅读合适的书籍也很重要。比起乱读一通自己还没有相关经验的书，或者乱读一通自己尚不具备相关预备知识的书，在合适的时间读合适的书会有更好的效果。太多的人在学习过程中过早地阅读了《Design Patterns》（中译本《设计模式：可复用面向对象软件的基础》，机械工业出版），那时像《Refactoring》（重构：改善既有代码的设计）这样的书可以针对模式提供更浅显的介绍。去“找人指导”吧，让他们对你下一步该读哪本书提些建议。时间选择会极大地影响你对一本书的体验。

行动指南

创建一个文本文件，可以将它纳入版本控制中。敲出你正在阅读的所有书籍。这就是你的“阅读列表”，也是这一模式的最简单实现。现在，你需要做的就是保持这份文件实时更新。

注1：*http://ravimohan.blogspot.com/2005/08/apprenticeship-pattern.html.*

注2：*http://www.industriallogic.com/papers/khdraft.pdf.*

参考模式

“找人指导”（第4章），“同道中人”（第4章），“分享所学”（第4章），“入门语言”（第2章）。

坚持阅读

即使每两个月读一本编程相关的好书，也就是每周35页左右，
用不了多久，你就会对我们的行业有深入的理解，
并使自己不同于周围的人。
——Steve McConnell，《Code Complete》

情景分析

你已经用“释放激情”打开了很多很多扇门。

问题描述

虽然精通了“入门语言”，你尚未搞懂的更深入、更基本的概念仍在源源不断地出现。

解决方法

重视你对学习的如饥似渴，消化掉尽可能多的文字。在你构建“阅读列表”的过程中，要更加重视书籍，而不是博客。

在“漫漫长路”上，应该有一段时间你拥有（或利用）机会阅读大量的书籍。对Dave来说，这段时间就是2002—2003年，在他开始编程的几年后，而且正是他的入门语言，Perl，达到稳定水平之时。这段时间是在公共交通工具上完成的：Dave每天有90分钟的时间在火车上，想读什么就读什么。他已经专注到在下车之后走一英里到达他格子间的路上也在继续读书。如果跟“找人指导”相结合，并与“同道中人”经常交流，将自己沉浸到本领域的经典名著和第一手资料中，可以带来无与伦比的学习体验。

这一埋头阅读的过程也应该包含对学术界大量知识库的探索。读一些偶尔翻到的研究论文可以拓宽你的思路，与计算机科学发展的前沿保持联系，而且提供了富有挑战性的新思想的源泉。试着实现这些思想，你的工具集将得以扩充，远在新的算法、数据结构和设计模式成为主流之前，便将它们囊入彀中。

行动指南

通过阅读本书，你已经开始运用这一模式了。诀窍在于，读完本书之后要继续保持这种阅读的动力。现在就确定接下来要读哪一本。先把它买来或借来，等读完这本书的时候，就可以马上转到下一本了。

另外，任何时候都应该随身携带一本薄书，利用每天当中的那一点点空余时间（比如在列车上，或者排队等待的时候）来学习。

参考模式

“找人指导”（第4章），“同道中人”（第4章），“阅读列表”（第6章），“漫漫长路”（第3章），“释放激情”（第2章），“入门语言”（第2章）。

钻研名著

找到自己专业或兴趣领域中的优秀作品——
所有最好的书籍、文章或演讲，
然后孜孜不倦地钻研这些作品。
——Joshua Kerievsky在“Knowledge Hydrant:
A Pattern Language for Study Groups”
（知识消防栓：面向学习小组的模式语言）中提到[注3]

情景分析

你是自学成才的，接受的是主张技能重于理论的高度实用型教育。

注3：*http://www.industriallogic.com/papers/khdraft.pdf.*

问题描述

与你合作的那些有经验的人们时不时地引用诸如“Brooks法则”[译注1]这样的概念，这些概念出自一些他们认为你（以及任何有自尊心的软件开发者）已经读过的书。

解决方法

“暴露无知”，向别人请教那些自己不知道的概念，以及它们出自何书。将这些书加入你的“阅读列表”。

有一次，Joshua Kerievsky去问Jerry Weinberg，出版的图书那么多，他是怎样跟上的。Jerry说：“很容易——我只读优秀作品”（《Refactoring to Patterns》（影印本《从重构到模式》，机械工业出版社引进），第33页）。通过“坚持阅读”和“且行且思”，你最终就能像Jerry那样，“只读最优秀的”。当你拿起一本书，想到的第一件事是它已经过时多久，这说明你正在读不恰当的书。成功的学徒常常关注“经久不衰的书”，然后通过上网和实验来学习这些知识如何演化。Dave读过的软件开发领域的第一本名著是《The Psychology of Computer Programming》（程序开发心理学），他当时惊诧于这本书对他是多么有用，尽管里面有很多关于穿孔卡片和有房间那么大的计算机的故事。这种阅读体验他至今记忆犹新。为使自己在漫漫长路上一直沿着正确的方向前进，这些经典名著所体现的智慧都是至关重要的信息。

关注于经典作品也有风险：你对它们投入过多的精力，而完全不顾那些能提高日常技能的、更加注重实效的知识和信息。在你的阅读列表中，要确保经典名著和现代的、更注重实效的图书和文章混合出现。

译注1：Brooks法则，Fred Brooks1975年在他的《The Mythical Man-Month》（人月神话）一书中提出：向进度落后的项目中增加人手，只会使进度更加落后。Brooks还打过另外一个有趣的比方：一个女人可以用九个月时间生产一名婴儿，但九个女人却不能用一个月时间生产一名婴儿。

行动指南

你的书堆中最古老的书是哪一本？首先把这本书读一读。下次当你浏览其他开发者的藏书时，注意最古老的那本，并请教那个人，为什么她仍然保留它。

参考模式

“暴露无知”（第2章），“坚持阅读”（第6章），“阅读列表”（第6章），“且行且思”（第5章），“漫漫长路”（第3章）。

深入挖掘

在实践中，算法问题不会在大项目刚开始的时候就出现。
然后，当突然之间程序员不知道如何继续下去或者
目前的程序显然不恰当时，它们就作为子问题出现了。

——Steven S. Skiena，《The Algorithm Design Manual》（算法设计手册）

情景分析

在你生活的世界上，有严格的项目最终期限，以及使用多种工具的复杂软件项目。老板还不能奢侈到雇用足够多的专家来填充每一种角色。对每种工具，你只学习到足以完成今天的工作。你选择一些跟手头工作所使用的语言和库有关的教程。你没有花时间去理解问题就做出了决策，并复制了工具中附带的玩具示例。这套方法是有效的，你可以靠它们做任何东西。你获得了迅速投入到新技术中并很快拿出解决方案的能力。你只学到足以让你那一部分系统运行起来的那一部分技术，然后依赖团队中的其他成员去学习其他部分。比如，你是一名服务器端的Java开发者，因此对如何构建用户界面知之甚少。

问题描述

你在维护自己编写的代码时不断遇到困难，因为你所仿效的教程走了一些捷径，并简化了复杂问题。每当出现一个棘手的bug，或者要做的工

作需要更深入的知识时，你发现对上千种工具的肤浅知识只是让自己不断挣扎。人们常常责怪你，说你的履历有误导性，因为，“扩展现有Web服务所需的几周时间”和“维护具有互操作性和高度可伸缩性的企业级系统时，对其中固有问题的深入了解”对你来说似乎没有区别。更糟糕的是，由于知识非常肤浅，你甚至不知道自己知道得很少，直到出现一些考验你水平的人或事。

解决方法

学会深入挖掘一些工具、技术和技艺。对知识的学习要达到“知其所以然”的程度。深度意味着要理解导致一种设计的推动力，而不仅仅是设计的细节。例如，它意味着你要理解类型理论（或至少*http://c2.com/cgi/wiki?TypingQuadrant*上面的类型象限所提供的简化版本），而不只是鹦鹉学舌似地重复着从别人那里听来的东西。

就像我们的一位前同事（Ravi Mohan，私人交流）发现的：

> 比起“派生Thread或实现Runnable”，对各种不同的并发形式（及其局限性）的了解是更加有用的知识。

你拥有深厚知识的那些领域能增强你的自信，并在决定如何运用“打扫地面”模式时为你提供指导，因为在你加入新团队时，这些领域可以指出项目中哪一块是你能尽快交付价值的。更重要的是，依靠这些深入的知识，你会获得一些力量，然后借助这种力量去尝试新领域的东西。你永远可以跟自己说：“如果我掌握了EJB，那么我也能处理元类（metaclass）。”

深入挖掘技术的另一好处是：对自己构建的系统，你可以真正解释其内部机制。面试的时候，这样的理解就能把你跟那些无法将自己所做的软件描述清楚的人区分开来，描述不清楚正是因为他们只理解很小的一部分。当你成为团队的一部分，对这一模式的运用会将那些胡乱堆砌碎石的人跟那些建造大教堂的人区分开来。对于前者，《Pragmatic Programmers》（程序员修炼之道——从小工到专家）一书中称之为

“靠巧合编程”（programming by coincidence），Steve McConnell则称之为“货物崇拜软件工程”（cargo cult software engineering）。

我们怎样识别教堂建造者呢？他们就是团队中那些去做调试、反编译和反向工程的人，那些阅读所用技术的规格说明、RFC或标准的人。做这些事情的人们已经拥有了新的视角，对支持他们的工具有了纯熟的理解。

这种视角的变换包括：愿意在一个系统中从上到下层层跟踪某个问题；愿意花时间弄清能够解释这一切的知识。比如，从单核处理器的笔记本电脑换到多核的电脑，可能影响到Java并发测试的行为。有些人只是耸耸肩，接受测试的行为将变得不可预知的这一事实。也有些人会跟踪这一问题，经由并发库、Java存储模型、物理硬件规格，直到CPU级别。

你需要熟悉的工具包括调试器（如GDB、PDB和RDB），它们让你窥探到运行中的程序内部；线级（wire-level）调试器（如Wireshark），它们让你看到网络流量；还有阅读规格说明书的意愿。除了读代码，你还能阅读规格说明书，这就意味着在你面前没有什么是秘密。对正在使用的库，你能提出有难度的问题，而且如果不喜欢找到的答案，你也有能力自己重新实现它，或者转向更符合标准的实现。

使用该模式的一种方法是从第一手资料中获取信息。这意味着如果下次有人跟你谈论表象化状态传变（Representation State Transfer）——说REST知道的人更多一些，你应该把这作为阅读Roy Fielding的博士论文的理由，正是Roy Fielding在他的博士论文里定义了这一概念。考虑撰写一篇博客来澄清或分享自己学到的东西，同时也鼓励其他人去阅读原始文档。

不要简单地记住别人说过的一句话，这句话可能是从一本书里引用的，那本书是解释一篇文章里的，那篇文章又提到了一个Wikipedia页面，而这个页面最终才链接到原始的IETF（Internet Engineering Task Force，因特网工程任务组）的“征求意见”（Request for Comment）文档。要真正理解任何思想，你都需要重建它第一次被表达时的上下

文。这样，你就可以理清经历了这么多中间人而保留下来的思想的精髓。

找出是谁第一次提出了那种思想，弄明白他们当初想要解决的问题。这样的上下文通常在思想四处传播的过程中由于各种转译而丢失了。有时你会发现类似的新思想在很久之前就被否决了，常常因为好的理由——但很多人早就忘了这一点，因为原始的上下文已经丢失了。你会一次又一次地发现，比起好多人年复一年选择性地相互引用，思想的原始来源是更好的老师。不管怎样，对一种你认为有用的思想，跟踪它的传承路线是一次重要的练习，而且是一种会让你在今后学习新事物时获益良多的好习惯。

阅读教程的时候，不要去寻找可以复制的代码，而应该寻找可用于放置新知识的思想结构。你的目标应该是理解某个概念的历史上下文以及它是否是另一种思想的特例。问自己，在你学习的知识背后，是否隐含着一种更基本的计算机科学概念，在你采用的实现中存在着怎样的权衡与取舍。具备了这种更深入的知识，当遇到问题的时候，你会比原来的教程走得更远。

例如，人们常常在使用正则表达式（regular expression）的时候遇到麻烦，因为他们只求肤浅的理解。你可以连续几年，甚至几十年平平安安，不需要真正理解确定性有限自动机（Deterministic Finite Automation）与非确定性有限自动机（Nondeterministic Finite Automation）之间的区别。然后，突然有一天你的wiki不工作了。结果证明，如果你用的正则表达式引擎是递归实现的，在遇到需要回溯的特定输入时，它会运行很长很长的一段时间，最后抛出一个StackOverflowException。Ade曾经费了很大力气才发现这个问题，幸好这只发生在他的玩具wiki的实现上，而非产品环境中。

有了对深入理解技术和工具的重视，你还需要当心，不要一不小心就变成了知识面狭窄的专家。你的目标是：不影响自己对软件开发各方面相对重要性的基本观点，在这一前提下，让自己获得足以解决任何问题的专业化知识。

尝试运用这一模式，你会发现获取深入的知识并非易事。这也解释了为何大多数人身上用于支撑软件开发的计算机科学知识都是又窄又浅。比起亲自付出额外的努力去获取知识，依赖其他人的基础知识更加容易，而且常常更有利可图。你可以跟自己说：“大不了到需要的时候再学它嘛”；然而，当那一天到来时，你将需要在周末之前明白一切，而学习所有的预备知识就需要一个月。

如果只拥有表面知识，另一种可能的后果是：你会永远意识不到自己正尝试解决的问题要么已经有了众所周知的解决方案，要么根本是不可能的（在后一种情况下，会有大量的学术论文讨论它为什么不可能，以及如何将它重新定义成一个可解决的问题）。如果浅尝辄止，你就了解不到自己所不知的东西；而不知道自己的知识边界，你也无法发现新的东西。洞穿一个问题所有层面的过程常会揭示一些来自计算机科学的基础概念。虽然计算机科学家的工作看起来不切实际，但那些能将最先进的理论运用到现实问题中的人将有能力做出其他人觉得不可思议的事情。选择一种不同的算法或数据结构，一个原本运行几个月的批处理任务将变成一件在用户松开鼠标按钮之前就已经结束了的事情。只知道List、Set和HashMap的人不太可能意识到他需要用Trie来解决自己的问题。相反，他只会觉得像最长前缀匹配（longest-prefix matching）这样的问题难得无法想象，然后要么放弃，要么去问可不可以降低这项特性的优先级。

如果有规律地运用这一模式，你会变成一个真正理解工具如何工作的人。你将不再只是把一小段一小段的代码粘合起来，然后依赖其他人的魔法去完成困难的工作。要知道这样的理解能把你跟与你共事的大多数程序员区分开来，并使你成为解决最困难任务时的必然选择。结果，你最有可能要么完全失败，要么出色地成功。另外，不要让这样的知识把你变得自负。相反，要继续寻找机会“只求最差”。挑战自己，基于这些基础部件组装有用的工具，而不只满足于拆卸它们的能力。

行动指南

找来RFC 2616读一读，它描述了HTTP1.1；还有RFC 707，它描述了

1976年1月时远程过程调用（Remote Procedure Call）技术的发展水平。有了对HTTP的深入了解，试着实现基于RFC 707的客户端和服务器程序。在你觉得对RFC 707的编者们所做的取舍有了较深的理解之后，考察一个基于同样思想的现代开源实现，如支撑了Facebook的Apache Thrift框架。然后，有前面这么多知识做铺垫，借着这一优势写一篇博客，描述一下前面三十年我们在远程过程调用和分布式系统方面的知识演进。

现在，去读一读Steve Vinoski关于RPC的文章。现在你对自己理解的深度有疑问吗？再写一篇博客，讲一讲自己的疑问以及当前的理解水平。

参考模式

“只求最差”（第4章），“打扫地面”（第4章）。

常用工具

泥沟（rut）是这样一种地方：你开动车轮时车却原地不动；
你取得的唯一进展就是给自己挖了条更深的沟。
石槽（groove）与此不同：车轮转动，
你也毫不费力地向前开走……陷在尝试过、检测过的方法上，
不考虑自己和周围的世界已经改变了，结果就是一条泥沟。
——Twyla Tharp，《The Creative Habit》（创造性的习惯）

情景分析

每一个项目都充满了需要学习的新东西——新的团队成员，新的角色，新的业务领域，新的方法，还有新的技术。

问题描述

面对这一切，必须有一样东西是不变的，否则你还不如去做研究算了。怎样向客户提供一些保障呢？当你告诉客户交付某种特性需要多少多少时间的时候，客户需要对你的交付能力有一些信心基础。

解决方法

找出一组常用的工具并关注它们。最好这些工具都是你不再需要阅读文档的——要么你心里知道所有最佳的使用方法、难点问题和FAQ，要么你已经把它们写在了博客、wiki或者你选择的任何“记录所学”的地方。有了这些知识，你就能对工作中的特定部分提供可靠的时间估算，从而将风险限制在新的、未探索过的领域上。

仅因为这些工具是你常用的并不意味着你应该常向别人推荐它们。有时，可用于完成工作的最佳工具并不是你最熟悉的工具。此时，你必须决定自己的生产率重要还是团队的生产率重要。你对Struts熟悉得不能再熟悉了，但这并不说明它一定好用。

然而，还是会有些工具，你一直带着它们从一个项目转到另一个项目。你之所以比接下来将要面试的这个人更有成效，这些工具就是一部分原因。如果遇到问题，你已经知道到哪里寻找答案。你明白这些工具所能解决的问题、所能引起的问题。因此，你也清楚什么时候不要使用它们，这跟知晓什么时候最适合运用它们同样重要。

久而久之，这一小撮工具会让你越来越觉得舒服。好的一面是带来了更高的生产率，但也有风险。如果不小心，你会开始把自己的常用工具视为“金锤子”，认为它们能解决所有问题。还有一种风险是，你会成为这些工具方面的专家，以至于即使出现了更好的工具，你也无法放下它们。

当你需要扔掉工具箱中的大部分工具时，真正的挑战才算到来。有时你的工具会变得过时；也有时你发现已经有了更好的工具。在较少的情况下，由于对“技术发展水平”了然于胸，你会发明一些超越已知工具的东西，使之不再必需。

> 在一个不断发生剧烈变化的时代，坚持学习的人才是未来的继承者。博学的人们常常发现自己有能力生活在不复存在的世界中。
>
> ——Eric Hoffer，《Reflections on the Human Condition》（人类状况反思）

Ade很早就采用了名叫Subversion的集中式源码控制系统。随着Subversion更加流行，会有客户请Ade为他们的项目提供帮助，因为他是这方面的专家。尽管如此，Ade却从一开始就在关注分布式源码控制系统这一新品种的出现。[注4]当Subversion变得过时，它在Ade工具箱中的位置将早已被Git[译注2]或Mercurial[译注3]所取代。放弃熟悉而又好用的工具是一种让人痛苦的过程，但也是一种需要学会的技能。

我们可以保证：你在学徒期使用的工具到你变成熟练工时肯定会变得过时。最终，你所钟爱的所有工具都将变成垃圾。为了职业生涯的成功，你必须学会从容地获取或放弃一些熟练工具。如何安排支持这一目标的学习过程，是所有学徒在转变成熟练工的过程中所面对的挑战之一。

行动指南

写下自己的常用工具列表。如果少于五项，就着手去搜寻一些工具，以填补工具箱的空白。你可能只需找出一种已经在用但了解得还不够深的工具，也可能要寻找全新的工具。不管是哪一种，整理出一份学习这些工具的计划，并从今天开始实施它。

如果你已经有五种常用工具，认真考察它们。是否存在更好更强大的工具？你是否仍在死守着已经过时的工具？是否有新兴的工具正在威胁你工具箱中的物件，使之变得过时？如果你对上面任何一个问题的回答是肯定的，那么从今天就启动替换这些工具的过程。如果你需要安全的地方来试验新工具，那就运用“质脆玩具”模式吧。

参考模式

“质脆玩具”（第5章）。

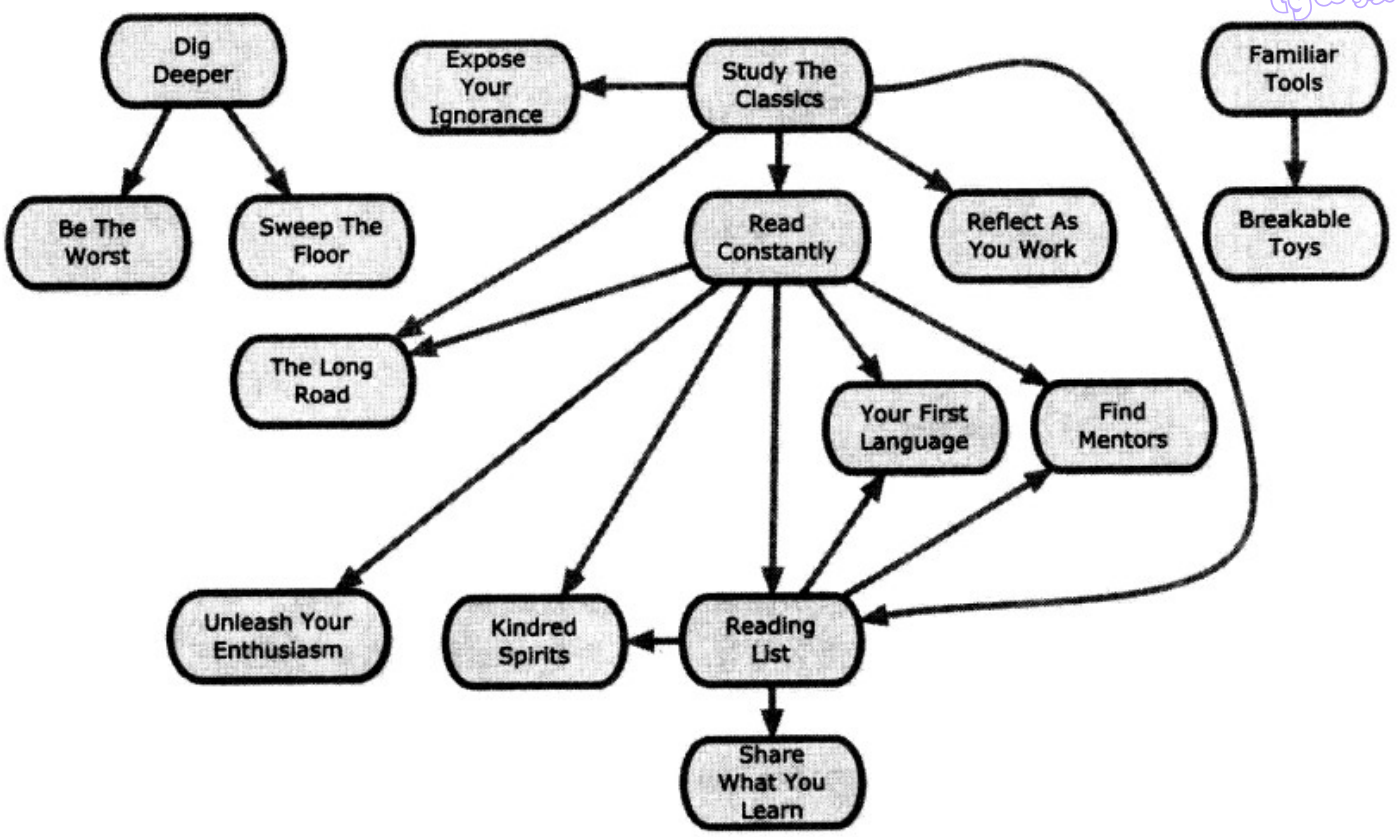
注4：Ade关于Subversion潜在替代品的研究：*http://delicious.com/ade/source-control-renaissance?setcount=100*。

译注2：最初由Linux之父Linus Torvalds设计并实现，用于管理Linux内核开发的免费分布式版本控制系统，见*http://git-scm.com/*。

译注3：Matt Mackall创建并主导开发的免费分布式源码控制管理工具，主要用Python语言实现，见*http://mercurial.selenic.com/*。

总结

在正式教育中，你可能已经习惯了一套为你准备好的课程，你一门一门学过来，很少考虑或从不考虑那些书是否是最值得阅读的好书，以及你是否在按最佳的顺序学习。现在，你需要成为不断自学过程中的主动参与者，并且成长为一名可以挥舞强大的工具来组织并收集信息的开发者。你可能并不通晓安排一套课程所需了解的一切，但你有能力去请教那些能为你提供建议的人，综合他们的智慧。学会享受学习的过程，在这个让我们永远保持警觉的不断变化着的技术田园中，这会让你受益匪浅。


Dig Deeper
Expose Your Ignorance
Study The Classics
Familiar Tools
Be The Worst
Sweep The Floor
Read Constantly
Reflect As You Work
Breakable Toys
The Long Road
Your First Language
Find Mentors
Unleash Your Enthusiasm
Kindred Spirits
Reading List
Share What You Learn

第7章

结束语

最让一名技师感到自豪的，是他熟练掌握的技能。
这也是简单的模仿不能一直让人满意的原因；
技能必须不断发展。
——Richard Sennett，《The Craftsman》（技师）

当我们开始写这本书的时候，我们只是想给有所追求、渴望技术的学徒们提供一些建议。然而，工艺式的方法中虽然存在着机遇，同时也有一些局限。

17至18世纪，Antonio Stradivari的工场制作出来的小提琴和大提琴被公认为全世界最精良的。他们常常卖到数百万美元；过去的300年里，人们多次尝试复制它们。然而，正如《The Craftsman》（[技师]，第75页）一书所说："像Antonio Stradivari和Guarneri del Gesù这样的师傅，他们的秘密实际上已随他们死去了。金山银山，加上无数的实验，都不能换回这些师傅的秘密。这些工场一定有某些特质阻止了知识的传递。"Stradivari晚年的时候，随着自己越来越老，他已经不能积极参与到学徒们的日常生活中了，他的工场制造出来的乐器，质量也在下降。由于他的工场是"以个体的非凡才能为中心"的，而Stradivari不能将自己的才能传递给学徒们，他的工场就随他而去了（《The Craftsman》，第76页）。

Stradivari的学徒包括他的两个儿子，他没有任何动机对他们隐瞒什么。据我们所知，他把自己所知的一切都教给了他们。或者更确切地讲，他把所有他认为很重要、认为学徒们必须知晓的东西都教给了他

们。他的失败，恰恰是因为那一点一滴的难以言传的知识，他甚至从没意识到那也是他技能的一部分。工场的衰落源于所有那些在那时显得并不重要，因而也没有记录下来的各种微妙联系，还有与学徒一起完成微不足道的任务时他所运用的难以言传的知识。

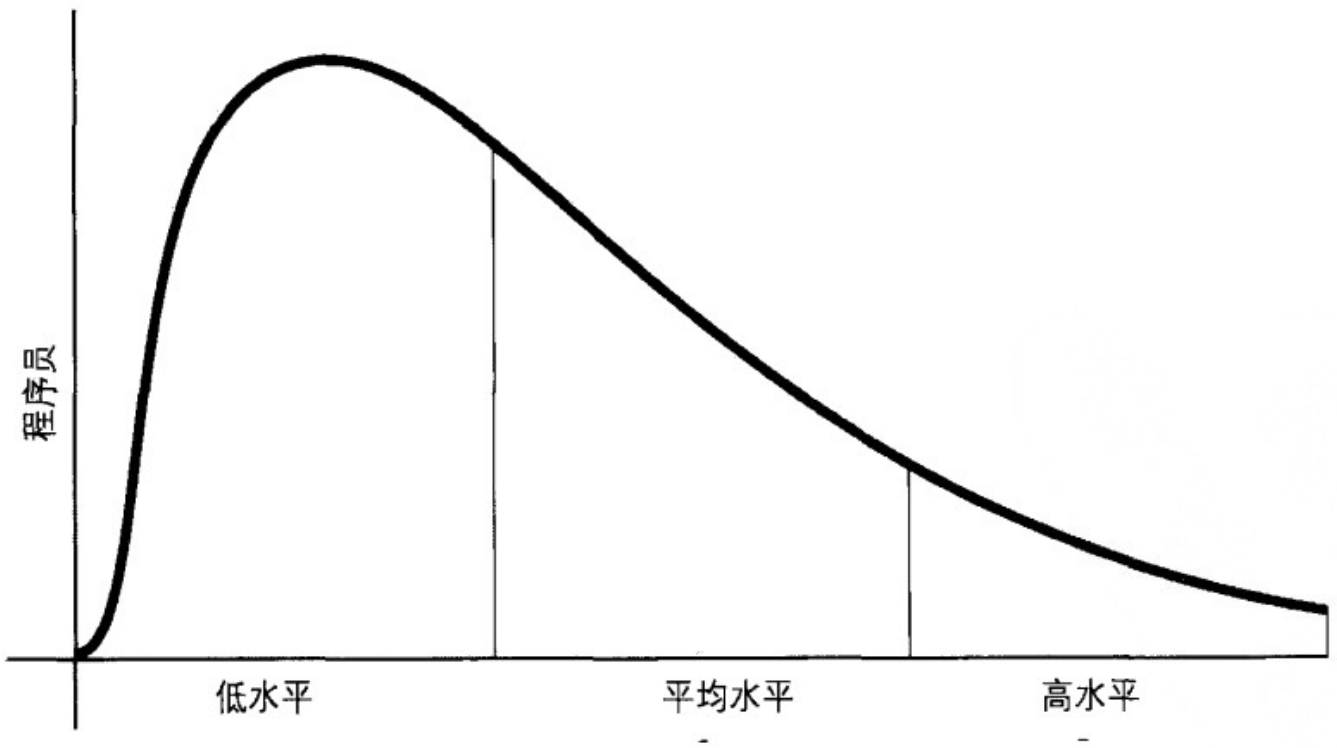
Stradivari没有在足够广泛的人群中分享自己的知识，他忘记把一些知识传授给客户，通过客户的要求让学生达到跟他的一样的标准。最终，他倾其一生所获得的经验也随之而去了。然而，我们应该历史地看待这样的失败。对Stradivari的学生们，音乐家们仍然说他们的工作“很棒，但只是很棒而已”（《The Craftsman》，第77页）。从技艺精湛的Stradivari身上，我们应该吸取的教训是：“大师们应该不厌其烦地表达自己，把存在于默然中的关键线索和动作整理出来”；我们还应该促使他们把那些难以言传的东西说明白。若没有那种热情到爱出风头，喜欢去促动别人的学徒，软件工艺仍将局限于一小撮才华横溢的开发者周围所形成的孤立品质中。

之所以说软件开发是一种工艺，正是因为我们现在对它理解得还不够深，不足以使之成为像科学或工程那样的系统化学科。虽然像软件工程学院（Software Engineering Institute）和敏捷联盟（Agile Alliance）这样的组织付出了大量的努力，我们所处的依然是一个个体技能对项目成功起最关键性决定作用的领域。我们使用“技能”一词，意思不只是你懂得多少计算机科学知识，你采用的开发过程多么高效，或者你有多少经验。我们指的是为交付可以工作的软件，你所需要的所有因素的总和。它包括但不限于你从本书的模式中学到的所有东西。

技能之所以如此重要，是因为我们对自己所做事情的了解还不足以将它写成一种可供别人直接运用并得到同样结果的格式。我们的客户希望软件项目能像科学实验一样可以重复。他们觉得最好能随便雇一支开发者团队就可以构建他们的系统，然后怡然自乐地以为只要团队具备最低的能力水平，他们就能得到自己想要的结果。事实情况是：客户不得不勉强组织起一支团队并期望他们能够胜任。他们希望，如果曾经有一支人数和经验水平都差不多的团队在过去能完成一件差不多的事情，那么今

天的团队或许也能完成。不幸的是，软件开发中的技能水平范围很广，在我们当中，最优秀的那些人可以稀松平常地做出大多数人认为不可能的事情。

此外，多数程序员都认为自己的水平超出平均。悲惨的事实是：由于下图所示的不规则的技能水平分布，大多数程序员实际上是低于平均水平的。这听起来有点反直觉，但你可以设想这样一个比方：现在我们两个人，Dave和Ade坐在桌子边上，然后Bill Gates过来加入我们，突然之间坐在桌子旁的“大多数”人的薪水都低于平均水平了。也就是说，在编程技能曲线上处于极远端的那些人倾斜了整个分布。把这一事实跟我们在“建立馈路”模式中讨论过的达克效应（Dunning-Kruger effect，也叫“无意识的无能”效应）结合起来，你就会明白为什么许多软件项目都以失败而告终。一边是我们所具备的技能水平，一边是解决当前问题所需要的技能水平，这两者经常不匹配。我们所能做的就是尽量改善开发方法。但任何过程，不管它多么敏捷多么精益，都无法告诉你你正在做的事是个NP完全问题，或者违反了CAP猜想。这些对你来说可能是比较晦涩的概念，同样，也会有程序员对正则表达式、HTTP或者UNIX感到晦涩。如果一个项目需要这些知识和技能才能成功，那么你就别无选择，必须具备它们。



当我们说一件事情是工艺，我们所要表达的一种意思是：它是一套高度重视技能的训练和传统。这包括学会、发扬并最终传承一种技能。我们相信，真正的精通闪耀在你传递自己的高超技能时对别人所形成的影响中。

Atul Gawande博士在他的《Better: A Surgeon's Notes on Performance》（更好：一位外科医师的性能外科笔记）一书中讲述了内科医师Ignac Semmelweis的故事（《Better》，第15页）。1847年，Semmelweis大夫研究发现："由于没有坚持洗手，或者洗得不够干净，在许多问题上该承担责任的是医生自己。"他把这归结为医院里导致产妇死亡的首要原因。他提出了一些简单的做法，比如要求医生在为不同的病人看病之间使用氯溶液洗手，就把死亡率从20%降到1%。然而，在获得这一显著结果的过程中，他疏远或得罪了所有的同事。最后情形恶化到同事们仅仅为了反对他而故意不洗手的地步。Semmelweis未能恰当地解释自己的观点或者说服别人，最终使他丢失自己的工作，更夺走了许多人的生命，直到20年后Joseph Lister发现了同样的结论。Gawande说得没错："Semmelweis是个天才，同时也是个疯子，而这使他成为一个失败的天才。"（《Better》，第17页）

同样，本书中的许多模式和思想都会招致抵触和反对。有一部分原因是总是有人可以从现状的维持中获得好处，这些人担心，如果现状得以改善，他们会失去一些东西。然而，我们需要从Semmelweis的失败中吸取教训。我们可以去清晰地解释自己的想法，说服人们试用我们的方法。我们还可以致力于营造一种欢迎积极变革的社区或组织，并寻找渴望精通技术的人们。

在软件开发中，我们并不确定哪些东西构成了精通的技艺，但我们知道哪些东西不属于其中的一部分。天赋、运气、财富、名声，这些并不能使你成为师傅。这些不是工艺的要素。涵盖软件开发方方面面的技能，以及传承这种技能并将技能训练推向前进的能力才是工艺的核心。

有一点可用于鉴别师傅，就是他们的学生最终都会在志气和成就方面超越他们。师傅们明白：绝对顺从其权威是危险的，因为"对卓越的追求

会引起组织寿命方面的问题，就像在Stradivari的工场里那样。完成高质量工作的经验包含在师傅自己难以言传的知识中，这意味着他的卓越品质不能传给下一代”（《The Craftsman》，第243页）。作为学徒，你应该力争比老师更好。而如果他们是好的老师，也应努力帮你达成这一目标。

师傅不是一种自己称呼自己的身份，因为人类的虚荣会限制人们自我评估的准确性。如果有人声称自己是师傅，那就让她证明一下。她指的是自己的工作吗？对一个技能水平比你高的人，评价其工作非常困难。你不能明白，她做的那些看来很容易的事情为什么实际那么困难。顶多，你可以说这个人技能水平比你高。但这并不足以证明她技艺精通。一个人走在了你的前面并不能证明她是个师傅。

她指的是自己的资质吗？但并没有一种“软件技师师傅”的资质证书。在达到技艺精通的道路上，一位候选者所能宣称的，顶多是她的同伴和那些她相信已经是师傅的人们都认为她迈过了这一步。这显然是个递归定义。只有那些已经是师傅的人可以说你是师傅，而在自己成为师傅之前，你又无法判断那些说你是师傅的人是否已经优秀到足以授予这种荣誉。不幸的是，没有其他方法能让一种新的工艺不断发展。所有的工艺都是从模糊的定义和混沌的标准开始，磕磕碰碰地来到这个世界上的。我们必须忍受这些，直到我们建立起可以清楚地证明从业者技能水平的社区和知识体系。

在那一天到来之前，发现精通技艺的最好方法只能是考察一个人和她的学徒们的工作质量。工作和学徒们的生活强调了技能方面，而不是师傅的天赋和运气。仅仅是天才并不能成就精通，但如果一个人可以培训他人，使之达到或超越自己的才华，那就是一种证据，证明此人有师傅的潜力。

本书的两位作者都不是师傅。顶多，我们都是熟练工，来分享前进道路上学到的东西。如果要给自己打个分，如果满分是10分，我们都想给自己打9分，但有时我们会遇到一些人，使我们一下子觉得满分应该扩充到100分。有许多从业者都是我们非常尊重的，然而软件开发工艺领域

仍然缺少师傅。这不是问题。软件是一门新的工艺——人们编写软件顶多有70年历史。所以，我们不应期望已经有了师傅级的软件工匠。

我们怎么能如此确定真正的师傅不存在呢？我们不能。即使我们还没发现师傅，他们也可能已经存在。缺少证据不能成为证明。[注1]然而，我们期望软件工艺师傅的存在可以使整个软件产业引起一些波动。这不仅因为他们的产品和工具更加精良，也因为他们的学生会更加优秀。从某个源头开始，将会涌现出一大批优秀的学徒和熟练工。这些学生将继续向外延伸；他们“比其他人更加快速地传道、学习和改变”的能力将拉开他们与我们这些人的距离。根据Gawande的“正向偏差”理论，这些师傅的工场会让人奇怪，但其成果将是无可否认的优越。仅仅复制他们的实践方法，你可以获得显著的提高，但要想跟上他们持续不断的进步，唯一的方法就是做这些师傅的学徒。

如果我们在本书一开始就告诉你大师并不存在，那样你就灰心了。现在你已经看到我们在短短几年中搜集出来的模式，也看到我们的技艺中有多少是你可以学习的，我们希望你把它们看作是一次机会，甚至一次挑战。我们希望自己的工作可以启发一些学徒去争论：现在……还没有大师。

注1：Absence of Evidence Is Evidence of Absence（缺少证据就是不在场的证明）：*http://www.overcomingbias.com/2007/08/absence-of-evid.html*.

模式列表

- 另辟蹊径（A Different Road）：你发现自己想要去的那个方向跟通往软件技能的道路并不相同。
- 只求最差（Be the Worst）：当快速超越了周围的每一个人，你的学习速度下降了。
- 质脆玩具（Breakable Toys）：你工作在一个不允许失败的环境中，却需要一个安全的环境来学习。
- 具体技能（Concrete Skills）：你想到一个优秀的团队中工作，然而你掌握的实用技能很少。
- 正视无知（Confront Your Ignorance）：你发现了自身知识中的许多漏洞，而你的工作需要你理解这些主题。
- 技重于艺（Craft over Art）：你需要向客户交付解决方案，你可以选择采用一种更简单且已证明有效的方案，也可以利用机会来创造一些新奇和美妙的东西。
- 建立馈路（Create Feedback Loops）：你不知道自己是否正遭受“无意识的无能”（unconscious incompetence）之苦。
- 深入挖掘（Dig Deeper）：你只拥有许多工具、技术和方法的肤浅认识，在尝试解决更困难问题的时候，不断地遇到障碍。

- 自定路线（Draw Your Own Map）：老板提供给你的职业道路全都不适合你。
- 提高带宽（Expand Your Bandwidth）：你对软件开发的理解较为狭隘，只关注日常工作中的低层次细节。
- 暴露无知（Expose Your Ignorance）：你发现了自身知识中的许多漏洞，担心人们会认为你根本不明白自己所做的东西。
- 常用工具（Familiar Tools）：你发现很难估算自己的工作，因为你的工具集和技术栈总是在快速变化。
- 找人指导（Find Mentors）：你发现自己花费大量的时间在发明轮子，然后不断遇到障碍，但却不清楚到哪里找人指导。
- 同道中人（Kindred Spirits）：你发现自己无人指导，束手无策，而且周围气氛看起来与自己的期望不一致。
- 学会失败（Learn How You Fail）：学习能力提高了你的成功几率，但失败和弱点依然存在。
- 培养激情（Nurture Your Passion）：你工作在一个不好的环境中，它扼杀你对软件工艺的激情。
- 不断实践（Practice, Practice, Practice）：日常编程活动不会给你通过犯错来学习的机会。
- 坚持阅读（Read Constantly）：虽然你快速掌握了许多东西，你尚未搞懂的更深入、更基本的概念却源源不断地出现。
- 阅读列表（Reading List）：需要阅读的书籍数量快速增加，你不可能读完它们。
- 记录所学（Record What You Learn）：你一遍又一遍地学到同样的经验。似乎没有一样能持续下来。
- 且行且思（Reflect as You Work）：随着你装进肚子里的工作年限和项目经历越来越多，你发现自己在等待着一种质变，使你神奇地变成“经验丰富”的开发者。

- 以退为进（Retreat into Competence）：当你发现了自己的大片无知领域时，你感觉自己要被淹没了。
- 密切交往（Rubbing Elbows）：你感觉有更高级的技术和方法而自己却抓不住。
- 分享所学（Share What You Learn）：周围的人学习起来没有你快，你感到失望了。
- 坚守阵地（Stay in the Trenches）：你获得一次提升的机会，组织想把你提升到一个不再编程的职位上。
- 钻研名著（Study the Classics）：你周围更有经验的人们不断地引用一些书中的概念，他们以为你已经读过那些书了。
- 持续动力（Sustainable Motivations）：你发现自己工作在一个令人失望的世界里，做着含糊不清的项目，面对着客户不断摇摆而且相互冲突的需求。
- 打扫地面（Sweep the Floor）：你是个缺少经验的开发者，需要赢得团队的信任。
- 深水区域（The Deep End）：你开始担心自己的职业并没有处在稳定水平，而是陷在了泥沟中。
- 漫漫长路（The Long Road）：你渴望成为一位软件师傅，而你的抱负同人们的期待不一致。
- 白色腰带（The White Belt）：你正在奋力学习，因为已有的经验似乎使新技能的学习更加困难了。
- 释放激情（Unleash Your Enthusiasm）：你发现自己为了适应团队而压抑自己对软件开发的兴奋和好奇。
- 使用源码（Use the Source）：如果你周围的人没有能力区分好代码和坏代码，你如何能认识到自己工作中哪些地方做得好呢？
- 使用头衔（Use Your Title）：当你在职业场合介绍自己的时候，

你都会觉得自己必须道歉或者专门解释一下自己技能水平和职位描述之间的差异。

- 入门语言（Your First Language）：你已熟悉了几门语言，但任何一门用得都不流畅。

附录B

一次学徒培训的号召

Dave Hoover

注解

这是我2007年8月22日发表在自己博客上的文章[注1]。这次行动号召（call to action）面向那些有能力雇佣学徒并建立学徒培训项目的人。即使你目前没有这种能力，我也希望你能把这些内容记在心里，伴随着你慢慢走过学徒生涯。

上周在2007敏捷大会上，我悄悄溜进了Bob大叔有关专业技能和职业精神的最后30分钟演讲中。当Bob大叔谈到工艺技能的时候，他主要在讲一些工艺方面的详尽细节，比如特定的实践方法和工具，但他还是有一张幻灯片是关于学徒训练的。他有些激动地抱怨现在的大多数高校，怪他们不能使毕业生掌握足够的技能，以便在上班的第一天交付高质量的软件。（更不用说数不胜数的从其他领域转向软件开发的人们，他们甚至连Bob所说的“不合格的计算机科学教育”都没接受到。）Bob认为我们需要把这些年轻人，也就是这些毕业生和新来的人，作为学徒带一带。他还声称最有效的学习环境应该是这样的：数量不多的学徒和数量更少的熟练工一起工作，而他们一起接受来自师傅的指导。听着Bob大叔的演讲，我的耳朵就像在听音乐，直到他询问观众席中有没有人在这样一种环境中工作。

我自豪地举起了手，然后我环顾四周，发现只有我一个人举手，这时我的心沉了一下。

注1：“Red Squirrel Reflections”（红松鼠沉思录），参见：*http://redsquirrel.com/cgi-bin/dave/craftsmanship/a.call.for.apprenticeship.html*.

在大会剩下的时间里，我因为发现Obtiva的软件工作室是如此稀奇的事情而感到一丝骄傲。同时我也在抵御一种沮丧和失望的感觉，因为我们这个行业能为毕业生和新来者提供的学徒机会这么少。我最大的失望是发现一些小公司（1～20名员工）完全由经验超级丰富的、世界一流的开发者、教练和和培训师组成。我能理解你们的压力，知道你们为什么只雇用那些已经小有名声而且有5年以上职业经验的人，但我认为，你们是在损害这个行业，因为你们不声不响地推诿了顺路带上几个学徒的责任。

当毕业生和新来者寻找第一份工作的时候，他们会去哪里呢？他们自然是去招收入门新人的地方。恰恰是这一点埋没了我们最大的一部分潜力，因为有潜力成就卓越的人们待在乏味臃肿官僚的开发组织中会逐渐丧失信心。设想一下，如果年轻的Nathaniel Talbott，由于缺乏经验和资质，不能做太多有趣的事情，只找到了一份“入门级”的职位，而不是成为RoleModel软件公司的第一名学徒，那结果会怎样？当然，很可能会有另外一个人为Ruby编写单元测试。而Nathaniel仍有可能成为不错的软件开发者。但我确信Nathaniel的学徒过程已经对我们的行业产生了影响，而我们的行业因为这种影响而变得更好。

学徒培训不只是雇佣入门级的新人那么简单。学徒培训是把一名学徒跟一名熟练工联结起来。并不是说他们一直结对编程，而是说熟练工要督导学徒的进步，对学徒来说，则是在物理距离很近的地方，有一名有经验的开发者，他可以随时向他请教。

此外，学徒也不一定都是入门级的新人。我们的第一批学徒大体上都有一到两年的工作经验。有些是未完成学业的在校生。有些最近才毕业。有一个人到年龄很大时才重启自己的IT生涯。学徒是那些愿意接受初级职位从而使学习机会最大化的人，而不是那些往经济收入最大化的职位上拼命攀爬，慢一分钟都不行的人。按我的经验，如果学徒拥有才干和正确的态度，其经济上的成功必然会伴随着学习的成功水到渠成。

请您考虑在自己的组织内建立学徒制度。我相信，面对着人才匮乏问题，学徒培训是我们这个行业最大的希望。

我很幸运地进入了优秀大学，在那里学习了很多的理论知识（那时还没有现在这么多理论）。然而，真正让这种经历闪光的是我参加过的学徒培训。一个毕了业的学生把我收入麾下。他没有直接教我什么，但他通过一些例子让我明白杰出的程序员如何思考。跟以前的课程学习相比，月复一月地在他旁边工作使我吸收了更多关于设计、编码和调试的实用知识。

后来，我又加入了伦敦的一家创业公司，在那里我参加了另外一种学徒培训。我的新老板向我指出：软件开发中人的因素和技术“同样”重要。他让我理解了软件方程式的业务一端，并教我杰出的开发者如何基于技术实力打造个人人际关系。

我从这两种完全不同的学徒培训中“毕业”，成为一名比刚起步时更有实力的开发者。因为这些经验，我现在是一名学徒培训的信徒。和师傅们一起工作是学习工艺的最佳途径。

——Dave Thomas，在McBreen的《Software Craftsmanship》一书中，xiv页

回顾Obtiva学徒训练项目的第一年

Dave Hoover

注解

这是我2008年3月23号发表在自己博客上的文章[注1]。在去年的行动号召之后，这些就是我所做的尝试，它包括以身作则的领导，还有把我尝试在Obtiva建立学徒项目时的一些得失分享给大家。自从写下这段回顾，在已有的两名学徒提升之后，我们又雇佣了两名新学徒。

一年之前，我不再做全职的现场咨询工作。我是2004年进入ThoughtWorks时开始做现场咨询工作的，然后伴随着我在Obtiva的第一个客户一直持续到后来。从2007年春天开始，我开始做一些定量的多日工作，但我的大部分时间都是在美国伊利诺伊州（Illinois）惠顿县（Wheaton）一个离家一英里的狭小办公室里度过的。启动Obtiva的“技能工作室”以及后来的学徒培训项目是一次冒险，但在大量的艰苦工作、频繁地犯错以及不完善的管理之后，我可以很自信地说：过去12个月的付出是值得的，未来也是光明的。让我试着解释一下我所谈论的事情，以及为什么我对我们第一年的成功如此自信。（Michael Hunger就这一话题询问了更多信息，顺便表示感谢。）

注1：“Red Squirrel Reflections”（红松鼠沉思录），参见：*http://redsquirrel.com/cgi-bin/dave/obtiva/apprenticeship.program.first.year.html*.

什么是“技能工作室”（Craftsmanship Studio）？或许我该问：“什么是Obtiva的‘技能工作室’？”因为我目前只有谈论这个的资格。首先，“技能”（Craftsmanship）在这里是指什么？我对这个问题的最佳回复是建议你去读Pete McBreen的《Software Craftsmanship》（软件工艺）的第三部分。作为一名自学出身而且拥有右脑化工作背景的程序员，工艺技能的概念很快引起了我的共鸣。这样，毫不奇怪，当我有机会在Obtiva内部建立自己的实践方法时，我就会按照Pete McBreen书中那些给我灵感的思想来行事。

其次，“工作室”在这里是指什么？字典告诉我们：“工作室”是“艺术家的工作场所”，或者“一个传授或学习艺术的组织机构”。这听起来是对的，也得到了事实证明：“计算机编程作为一门艺术”（Computer Programming as an Art），在我们这个领域的领军人物当中，长久以来一直都是牢不可破的主题。

以上观点基本概括了我们对“技能工作室”的设想：一个编程新手可以跟经验丰富的开发者密切合作，基于真实项目来学习软件开发工艺的地方。事实情况比我们的设想要混乱得多，有太多时间学徒们被孤立起来，或者一起工作但没有人照看。这些混乱可以归因于一个事实：我只是一名熟练工，而不是一名师傅级的工匠；因此，这一年里，在我（通过反复试验）学习项目管理、客户关系、能力规划和人员招聘的过程中，我犯过很多的错误。幸运的是，在那段时间我们有过大约50次的回顾反思，并将敏捷原则改编成一种每周都在持续进步的过程。

如果我们的“技能工作室”是一个新人和老员工工作、学习和进行指导的地方，那么学徒（apprentice）又是什么呢？在Pete的书中，第三部分可以快速地向你介绍学徒期（apprenticeship）的概念：

> 工艺模型的一个重要特征是：很难仅仅通过说教让别人学会一项技能。实际上，你必须在现实的条件下，在一个能提供反馈的、有经验的从业者的悉心关注下练习这项技能。

这描述了需求和关系，但现实中这些假设的新人又是谁呢？对于处在学徒期中的人，我是这样定义的：一个甚至不惜牺牲其他机会来最大限度

地获得学习机会的人。这常常意味着故意将自己置于“只求最差”这样的情景中，我刚加入ThhoughtWorks时就是这么做的。对Obtiva，这意味着我们注重的是潜力和态度，而不是资历背景。在短期内，这些人可以到别处赚更多的钱，但他们是在做一笔投资，来学习可以得到长期回报的东西。

我希望看到我们的学徒项目能够更加成熟，成为一种具有更好的反馈机制和里程碑的正式学徒训练过程。我仍可以说，虽然作为管理者我有很多缺点，但参加我们“技能工作室”的四名学徒都非常成功。这一年的经历进一步印证了我做学徒时学到的一点：学徒期完全是你自己安排造就的。我们有一位Perl Web开发者，到我们这里学习Ruby、Rails和Java，离开时为一家本地的对冲基金公司写了一个支持16核机器的多线程的Ruby/JRuby/Java/JNI应用。有一个人为了重启自己的职业来到我们这里，学习UNIX、MySQL、Perl、Ruby和Rails，现在为工作室最大的客户管理并开发电子商务系统。有一个人从一家本地的Rails血汗工厂来到我们这里，学习诸如Sphinx、rSpec、god、ActiveMerchant、CruiseControl.rb和Perl这样的技术，现在他正把Git引进团队，而且很快就会成为他的第三个Rails电子商务项目的主管了。还有一名网络管理员来到我们这里，后来很快交付多个不同的Rails项目，现在正在处理一个巨大而又混乱的通过rSpec管理的Rails代码库，比我知道的许多经验丰富的开发者做得还要好。

该把我们的成功归功于哪些因素呢？

- 共处一地：没有什么能打败面对面的团队合作。
- 结对编程：没有什么能打败肩并肩的开发。
- 测试驱动开发：没有什么能打败具有微型反馈回路的“乒乓编程”。
- 敏捷原则：根据我们坚持的原则，定期重新评估我们的现状，并相应地调整开发过程。
- 尊重客户：直接与“目标施主”（常常与“黄金属主”是同

一人）[译注1]一起工作，并使用他们的语言建立用户故事（user story）。

- 精良工具：我们使用具有超平板屏幕的Macs机器，并定期引入新技术来促进生产率。
- 努力工作：不管条件多么困难，我们都全力关注自己要交付的产品。
- 文化：没有什么能打败一个可以在停车场上打雪仗、让创造性的活力四处流淌的即兴团队。

所有这些加到一起就形成了不断增长的服务需求，我们正逐渐变得更善于管理它们，这些因素相互结合，让一切都按照可持续的节奏快步前进。这种节奏是关键的部件，因为这意味着我们同样拥有办公室之外的生活，为第二天的团队协作补充精力。

译注1：原作者使用的两个词分别是“Gold Donor”和“Gold Owner”。

在线资源

- 本书网址：*http://softwarecraftsmanship.oreilly.com/wiki*
- 本书英文版在O'Reilly Media网站上的目录页面：*http://oreilly.com/catalog/9780596518387/*
- 软件工艺邮件列表：*http://group.google.co.uk/group/software_craftmanship*
- 敏捷宣言：*http://agilemanifesto.org/*
- 软件工艺宣言：*http://manifesto.softwarecraftsmanship.org/*
- 软件工艺大会：*http://parlezuml.com/softwarecraftsmanship/*
- 软件工艺北美大会：*http://scna.softwarecraftsmanship.org/*
- Wikipedia上的“软件工艺”：*http://en.wikipedia.org/wiki/Software_Craftsmanship*
- 极限编程周二俱乐部：*http://www.xpdeveloper.net/*
- C2 wiki：*http://c2.com/cgi/wiki?FrontPage*
- Ade在C2 wiki上的页面：*http://c2.com/cgi/wiki?AdewaleOshineye*
- Dave在C2 wiki上的页面：*http://c2.com/cgi/wiki?DaveHoover*
- Bookshelved wiki：*http://bookshelved.org/cgi-bin/wiki.pl?FrontPage*
- Ade在Bookshelved wiki上的页面：*http://bookshelved.org/cgi-bin/wiki.pl?AdewaleOshineye*

- Dave在Bookshelved wiki上的页面：*http://bookshelved.org/cgi-bin/wiki.pl?DaveHoover*

参考文献

[Alexander] Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979.

[Alexander2] Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[Armour] Armour, Phillip G. *The Five Orders of Ignorance*. Communications of the ACM, October 2000.

[Beck] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[Beck2] Beck, Kent. *Test-Driven Development: By Example*. Addison-Wesley, 2000.

[Becker] Becker, Howard S. *A School Is a Lousy Place to Learn Anything In*. American Behavioral Scientist, September/October 1972.

[Bentley] Bentley, Jon. *Programming Pearls*. Addison-Wesley, 1999.

[Bentley2] Bentley, Jon. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1998.

[Brooks] Brooks, Frederick P. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[Brown] Brown, Jr., H. Jackson. *Life's Little Instruction Book*. Thomas Nelson, 2000.

[Coplien] Coplien, James, and Neil Harrison. *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004.

[Constantine] Constantine, Larry. *The Peopleware Papers: Notes on the Human Side of Software*. Prentice Hall, 2001.

[DeMarco] DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House Publishing, 1999.

[Dweck] Dweck, Carol S. *Mindset: The New Psychology of Success*. Ballantine Books, 2007.

[Dweck2] Dweck, Carol S. *Self-theories: Their Role in Motivation, Personality, and Development*. Psychology Press, 2000.

[Ericsson] Ericsson, K. Anders, Ralf Th. Krampe, and Clemens Tesch-Romer. *The Role of Deliberate Practice in the Acquisition of Expert Performance*. Psychological Review, 1993.

[Farleigh] Farleigh, John. *Fifteen Craftsmen on Their Crafts*. The Sylvan Press, 1945.

[Fowler] Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gamma] Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gawande] Gawande, Atul. *Better: A Surgeon's Notes on Performance*. Metropolitan Books, 2007.

[Graham] Graham, Paul. *Hackers & Painters: Big Ideas from the Computer Age*. O'Reilly Media, 2004.

[Highsmith] Highsmith, Jim. *Agile Software Development Ecosystems*. Addison-Wesley, 2002.

[Hoffer] Hoffer, Eric. *Reflections on the Human Condition*. Hopewell Publications, 2006.

[Hunt] Hunt, Andy. *Pragmatic Thinking and Learning: Refactor Your Wetware*. Pragmatic Bookshelf, 2008.

[Jeffries] Jeffries, Ron, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[Kerievsky] Kerievsky, Joshua. *Refactoring to Patterns*. Addison-Wesley, 2004.

[Kerth] Kerth, Norman L. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House Publishing, 2001.

[Knuth] Knuth, Donald. *Computer Programming as an Art*. Communications of the ACM, 1974.

[Kruger] Kruger, Justin, and David Dunning. *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments*. Journal of Personality and Social Psychology, 1999.

[Lammers] Lammers, Susan. *Programmers at Work: Interviews With 19 Programmers Who Shaped the Computer Industry*. Tempus Books, 1989.

[Lave] Lave, Jean, and Etienne Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, 1991.

[Leonard] Leonard, George. *Mastery: The Keys to Success and Long-Term Fulfillment*. Plume, 1992.

[Lewis] Lewis, C. S. *The Weight of Glory and Other Addresses*. HarperOne, 2001.

[McBreen] McBreen, Pete. *Software Craftsmanship: The New Imperative*. Addison-Wesley, 2001.

[McConnell] McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[Meyer] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[Peter] Peter, Laurence J, Raymond Hull, and Robert I Sutton. *The Peter Principle: Why Things Always Go Wrong*. HarperBusiness, 2009.

[Pirsig] Pirsig, Robert. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. Harper Perennial Modern Classics, 2008.

[Postrel] Postrel, Virginia. *The Future and Its Enemies: The Growing Conflict over Creativity, Enterprise and Progress*. Free Press, 1999.

[Rogers] Rogers, Carl, and Peter D Kramer. *On Becoming a Person: A Therapist's View of Psychotherapy*. Mariner Books, 1995.

[Sennet] Sennet, Richard. *The Craftsman*. Yale University Press, 2009.

[Skiena] Skiena, Steven S. *The Algorithm Design Manual*. Springer, 2008.

[Sudo] Sudo, Philip. *Zen Guitar*. Simon & Schuster, 1998.

[Surowiecki] Surowiecki, James. *The Wisdom of Crowds*. Anchor, 2005.

[Suzuki] Suzuki, Shunryu. *Zen Mind, Beginner's Mind*. Shambhala, 2006.

[Tharp] Tharp, Twyla, and Mark Reiter. *The Creative Habit: Learn It and Use It for Life*. Simon & Schuster, 2005.

[Thomas] Thomas, Dave, and Andy Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[Tractate] *Tractate Avot* (f (*http://www.jewishvirtuallibrary.org/jsource/Talmud/avot4.html*)).

[Vlissides] Vlissides, John M. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

[Wall] Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, 2000.

[Weick] Weick, Karl E, and Karlene H. Roberts. *Collective Mind in Organizations: Heedful Interrelating on Flight Decks*. Administrative Science Quarterly, 1993.

[Weinberg] Weinberg, Gerald M. *Becoming a Technical Leader: An Organic Problem-Solving Approach*. Dorset House Publishing, 1986.

[Weinberg2] Weinberg, Gerald M. *More Secrets of Consulting: The Consultant's Tool Kit*. Dorset House Publishing, 2001.

[Weinberg3] Weinberg, Gerald M. *The Psychology of Computer Programming: Silver Anniversary Edition*. Dorset House Publishing, 1998.

[Wetherell] Wetherell, Charles. *Etudes for Programmers*. Prentice Hall, 1978.

[Williams] Williams, Laurie. *Pair Programming Illuminated*. Addison-Wesley, 2002.

[Whitehead] Whitehead, Alfred North. *An Introduction to Mathematics*. BiblioLife, 2009.